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
```

```
class FibIter:
    def __init__(self):
        self._next = 0
        self._addend = 1
```

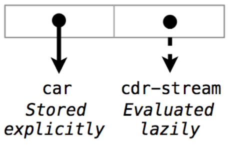
"Please don't reference these directly. They may change."

```
    def __next__(self):
        result = self._next
        self._addend, self._next = self._next, self._addend + self._next
        return result
```

```
>>> fibs = FibIter()
>>> [next(fibs) for _ in range(10)]
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
```

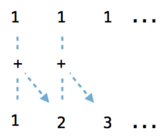
```
(car (cons 1 2)) -> 1
(cdr (cons 1 2)) -> 2
(car (cons 1 (/ 1 0))) -> ERROR
(cdr (cons 1 (/ 1 0))) -> ERROR

(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))
(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 (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)))))
```

[redacted]

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
```

```
class LetterIter:
    def __init__(self, start='a', end='e'):
        self.next_letter = start
        self.end = end

    def __next__(self):
        if self.next_letter >= self.end:
            raise StopIteration
        result = self.next_letter
        self.next_letter = chr(ord(result)+1)
        return result

class Letters:
    def __init__(self, start='a', end='e'):
        self.start = start
        self.end = end

    def __iter__(self):
        return LetterIter(self.start, self.end)

def letters_generator(next_letter, end):
    while next_letter < end:
        yield next_letter
        next_letter = chr(ord(next_letter)+1)
```

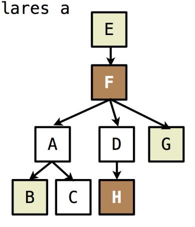
- A generator is an iterator backed by a generator function.
- Each time a generator function is called, it returns a generator.

```
>>> a_to_c = LetterIter('a', 'c')
>>> next(a_to_c)
'a'
>>> next(a_to_c)
'b'
>>> next(a_to_c)
Traceback (most recent call last):
    ...
StopIteration

>>> b_to_k = Letters('b', 'k')
>>> first_iterator =
b_to_k.__iter__()
>>> next(first_iterator)
'b'
>>> next(first_iterator)
'c'
>>> second_iterator = iter(b_to_k)
>>> second_iterator.__next__()
'b'
>>> first_iterator.__next__()
'd'

>>> for letter in
letters_generator('a', 'e'):
...     print(letter)
a
b
c
d
```

A simple fact expression in the Logic language declares a relation to be true.

Language Syntax:

- A relation is a Scheme list.
- A fact expression is a Scheme list of relations.



```
logic> (fact (parent delano herbert))
logic> (fact (parent abraham barack))
logic> (fact (parent abraham clinton))
logic> (fact (parent fillmore abraham))
logic> (fact (parent fillmore delano))
logic> (fact (parent fillmore grover))
logic> (fact (parent eisenhower fillmore))
```

Relations can contain relations in addition to atoms.

```
logic> (fact (dog (name abraham) (color white)))
logic> (fact (dog (name barack) (color tan)))
logic> (fact (dog (name clinton) (color white)))
logic> (fact (dog (name delano) (color white)))
logic> (fact (dog (name eisenhower) (color tan)))
logic> (fact (dog (name fillmore) (color brown)))
logic> (fact (dog (name grover) (color tan)))
logic> (fact (dog (name herbert) (color brown)))
```

Variables can refer to atoms or relations in queries.

```
logic> (query (parent abraham ?child))
Success!
child: barack
child: clinton

logic> (query (dog (name clinton) (color ?color)))
Success!
color: white

logic> (query (dog (name clinton) ?info))
Success!
info: (color white)
```

A fact can include multiple relations and variables as well:

(fact <conclusion> <hypothesis$_0$> <hypothesis$_1$> ... <hypothesis$_N$>)

Means <conclusion> is true if all <hypothesis$_K$> are true.

```
logic> (fact (child ?c ?p) (parent ?p ?c))

logic> (query (child herbert delano))
Success!

logic> (query (child eisenhower clinton))
Failure.
```

```
logic> (query (child ?child fillmore))
Success!
child: abraham
child: delano
child: grover
```

A fact is recursive if the same relation is mentioned in a hypothesis and the conclusion.

```
logic> (fact (ancestor ?a ?y) (parent ?a ?y))
logic> (fact (ancestor ?a ?y) (parent ?a ?z) (ancestor ?z ?y))

logic> (query (ancestor ?a herbert))
Success!
a: delano
a: fillmore
a: eisenhower
```

The Logic interpreter performs a search in the space of relations for each query to find a satisfying assignment.

```
(parent delano herbert)     ; (1), a simple fact
(ancestor delano herbert)   ; (2), from (1) and the 1st ancestor fact
(parent fillmore delano)    ; (3), a simple fact
(ancestor fillmore herbert) ; (4), from (2), (3), & the 2nd ancestor fact
```

Two lists append to form a third list if:

- The first list is empty and the second and third are the same.
- The rest of first and second append to form the rest of third.

```
logic> (fact (append-to-form () ?x ?x))
logic> (fact (append-to-form (?a . ?r) ?y (?a . ?z))
             (append-to-form ?r ?y ?z))
```

The basic operation of the Logic interpreter is to attempt to *unify* two relations.

Unification is finding an assignment to variables that makes two relations the same.

```
( (a  b) c  (a  b) )
(    ?x  c     ?x  )   ▷  True, {x: (a b)}

( (a  b) c  (a  b) )
( (a ?y) ?z (a  b) )   ▷  True, {y: b, z: c}

( (a  b) c  (a  b) )
(    ?x ?x     ?x  )   ▷  False
```

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)          > (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>)
```

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 recursive lists.
• A (recursive) 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.

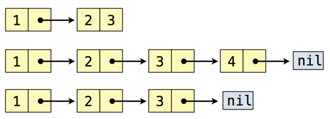
```
> (car '(a b c))
a
> (cdr '(a b c))
(b c)
```

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)
```



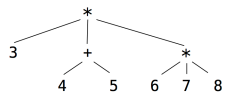
```python
class Pair:
    """A Pair has first and second attributes.

    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

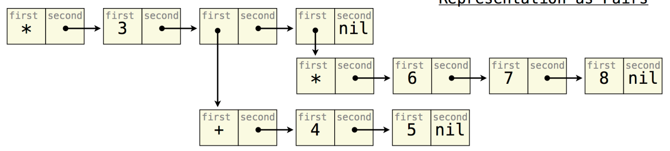
Calculator Expression

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

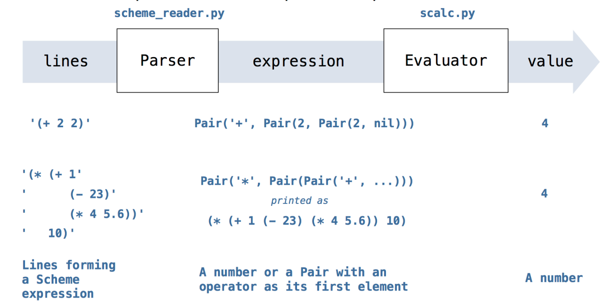
Expression Tree



Representation as Pairs



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

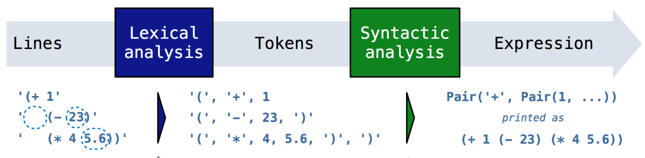


A Scheme list is written as elements in parentheses:

(<element$_0$> <element$_1$> ... <element$_n$>)

A recursive Scheme list

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.

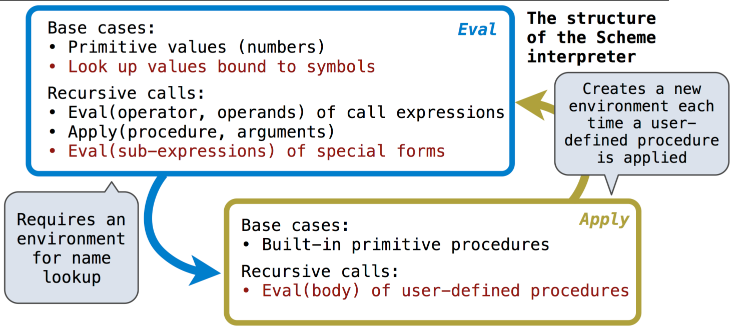


• Iterative process
• Checks for malformed tokens
• Determines types of tokens
• Processes one line at a time

• 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



Eval
Apply
Creates a new environment each time a user-defined procedure is applied
Requires an environment for name lookup

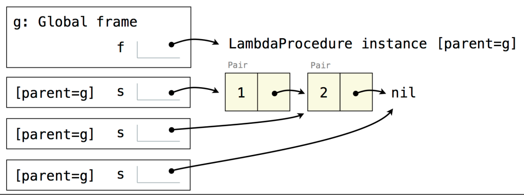
The structure of the Scheme interpreter

Eval
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

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

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