CS 61A Final Exam Study Guide – Page 1

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.

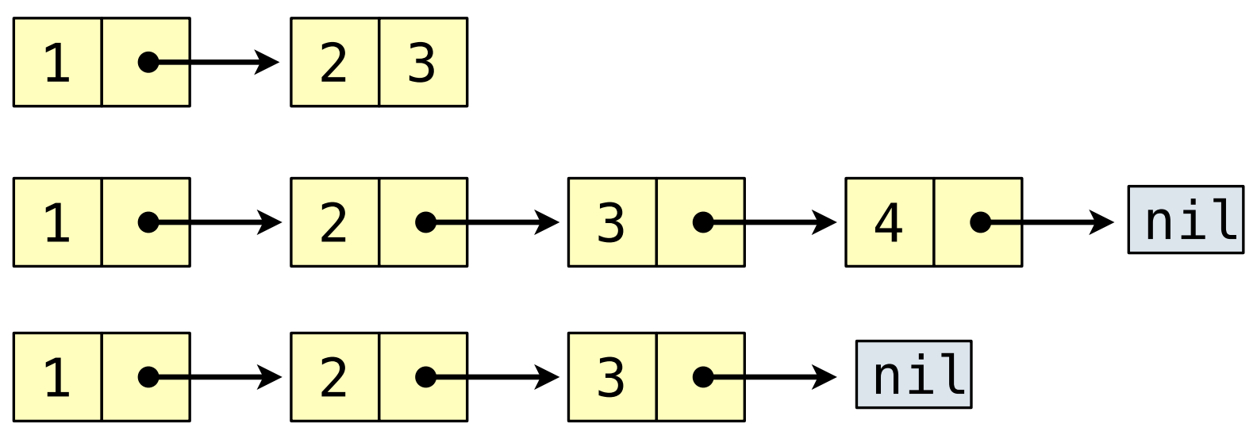
```
> (car '(a b c))
a
> (cdr '(a b c))
(b c)
```

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)
```

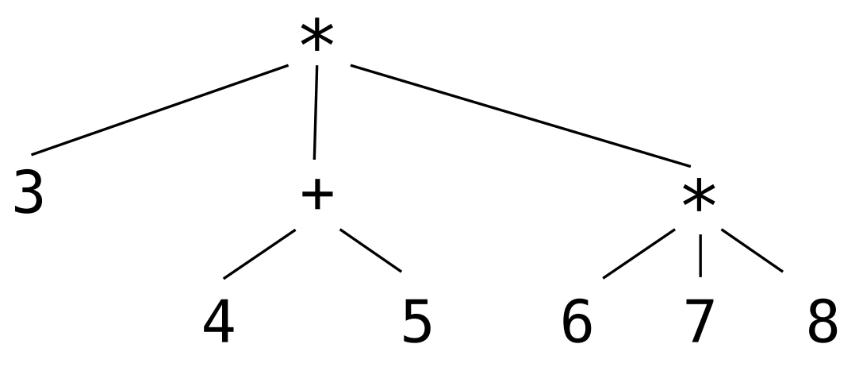


```python
class Pair:
    """A Pair has first and second attributes.

    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)
```

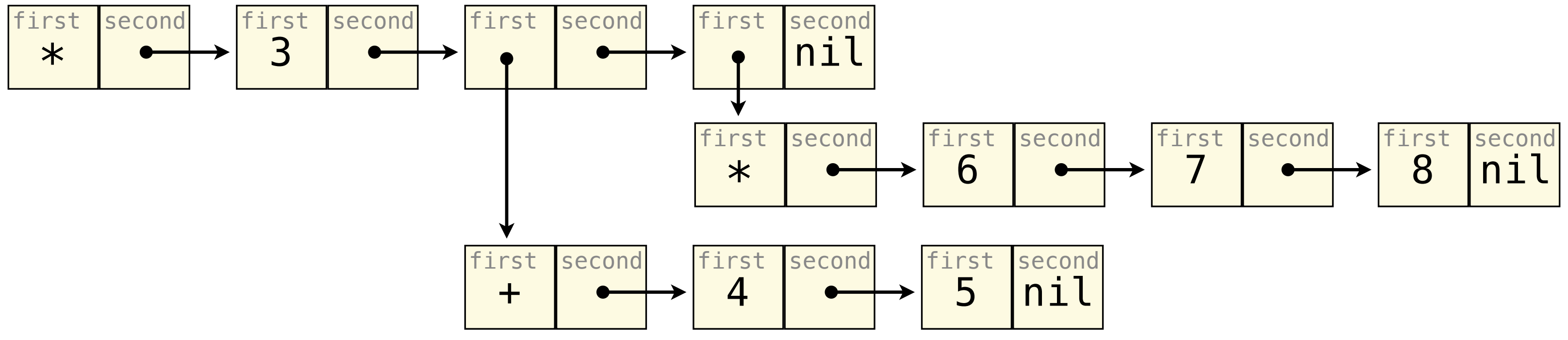
Scheme expression

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

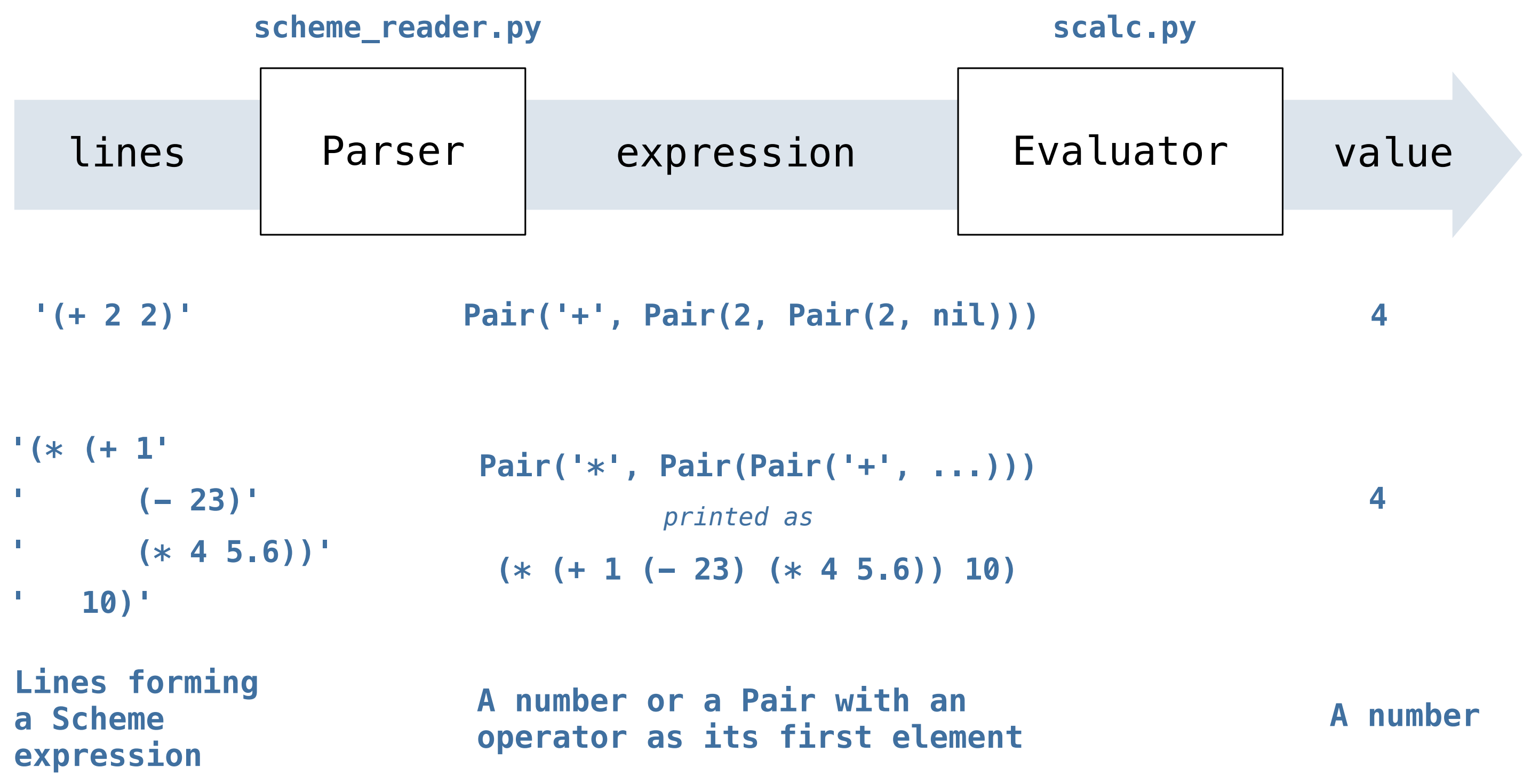
Expression Tree



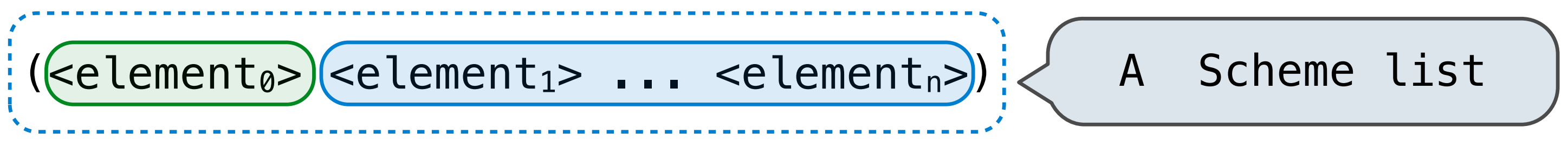
Representation as Pairs



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



A Scheme list is written as elements in parentheses:



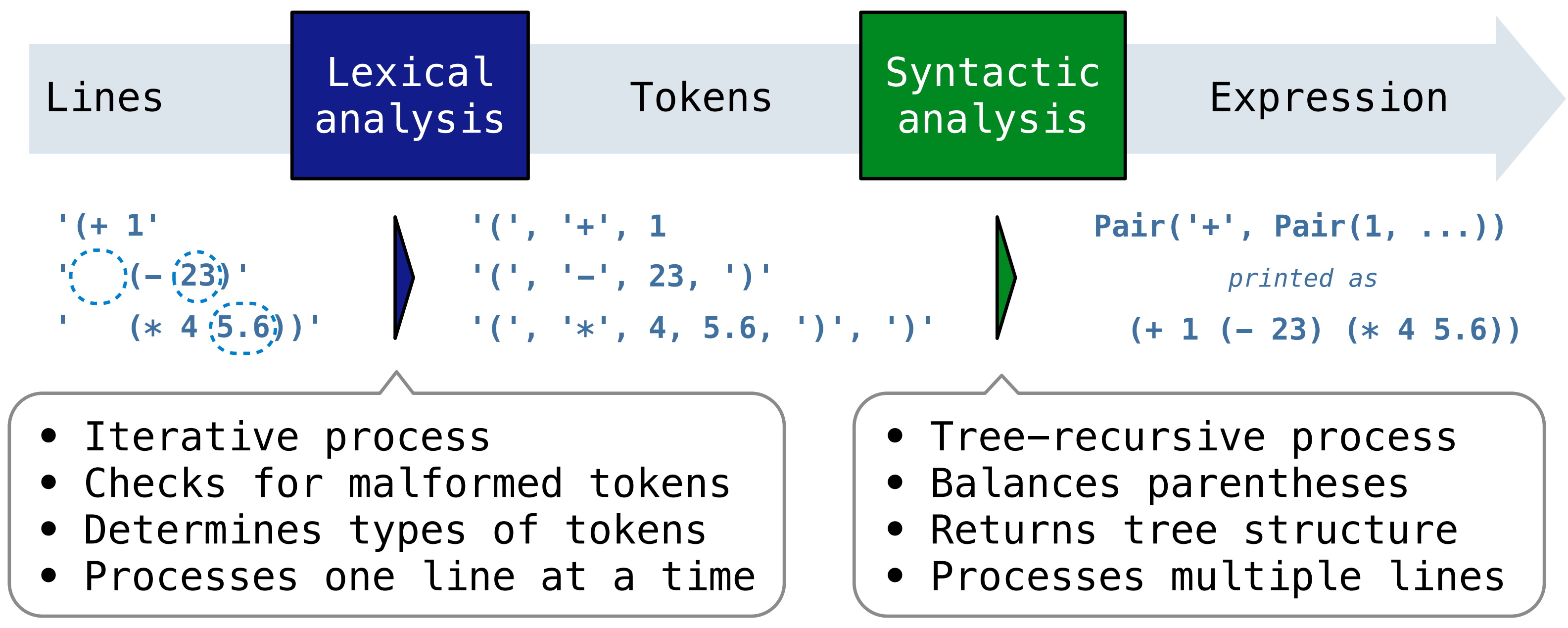
Each <element> can be a combination or atom (primitive).

(+ (* 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.



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

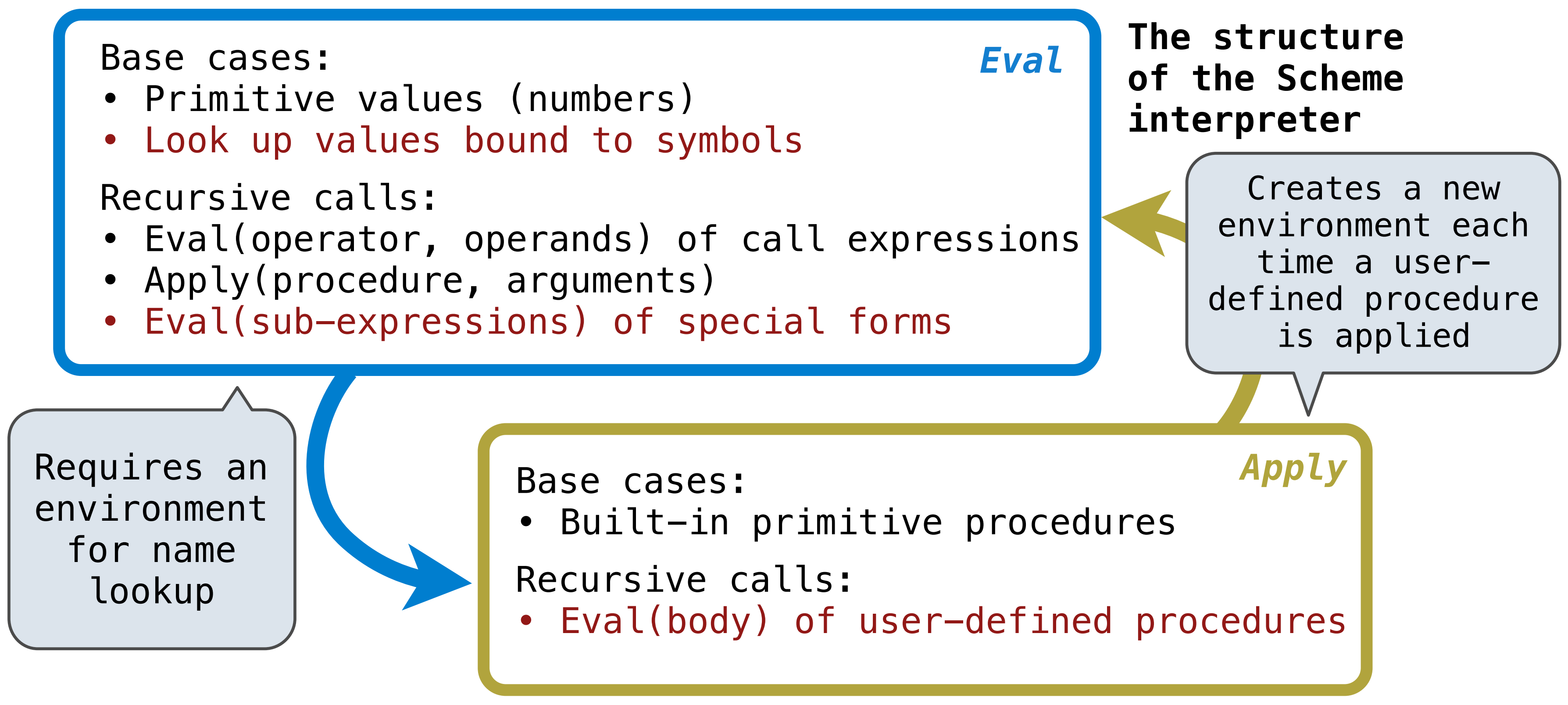
- Tree-recursive process
- Balances parentheses
- Returns tree structure
- Processes multiple lines

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

The structure of the Scheme interpreter



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

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

Requires an environment for name lookup

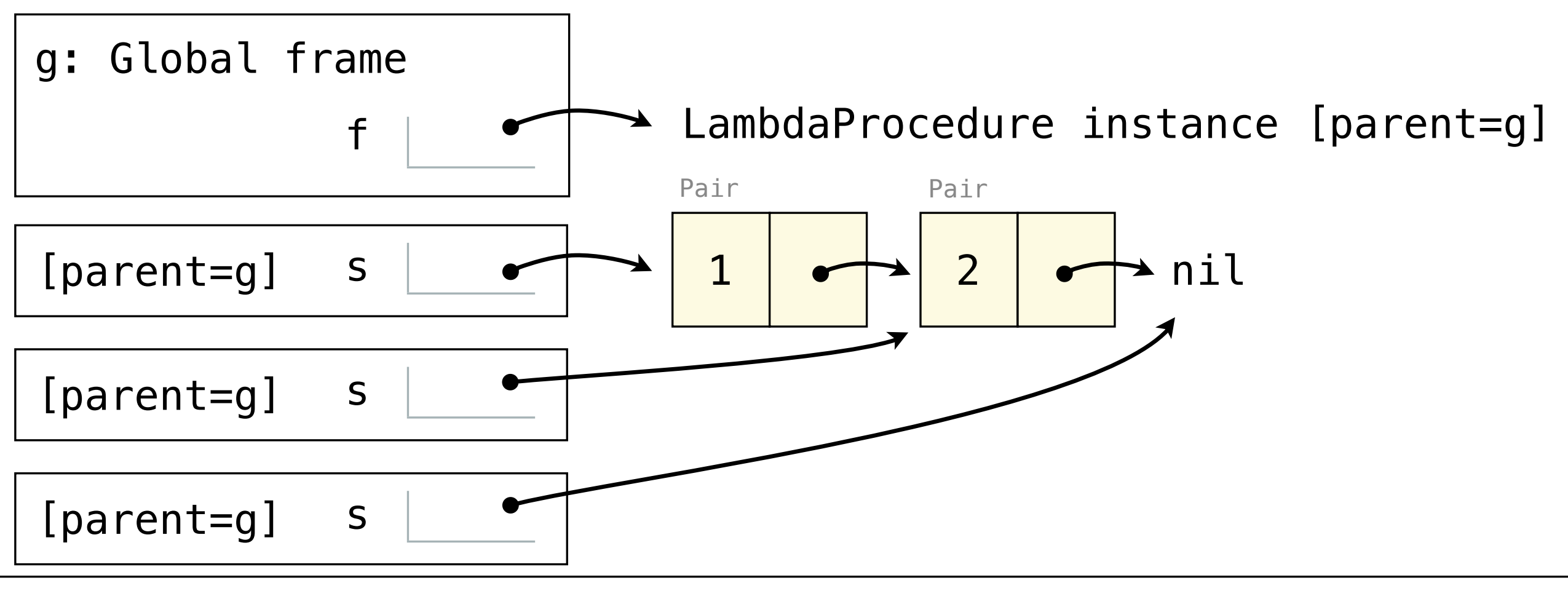
Base cases:
- Built-in primitive procedures

Recursive calls:
- Eval(body) of user-defined procedures

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

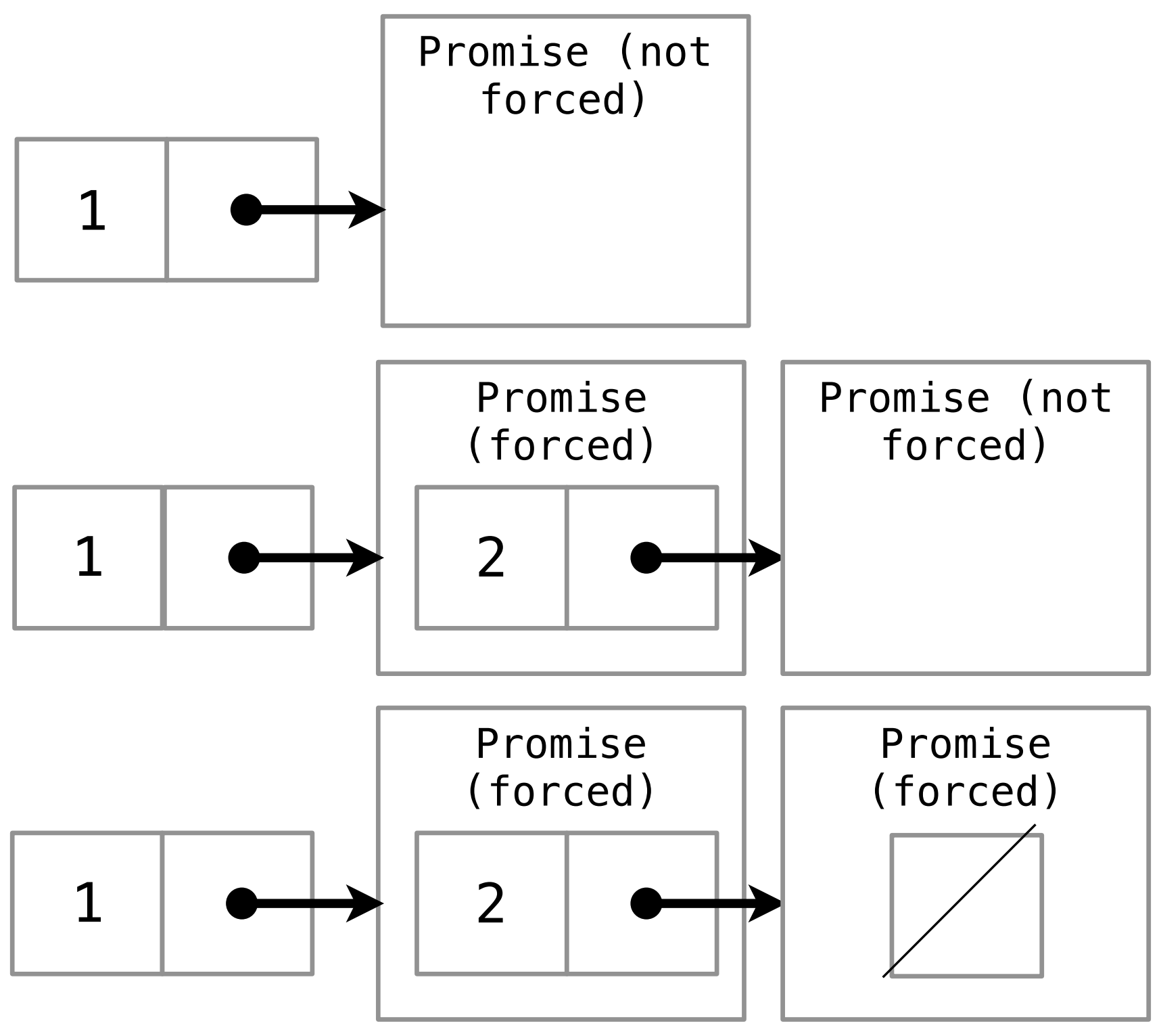
```
(define (length-tail s)
  (define (length-iter s n)
    (if (null? s) n
      (length-iter (cdr s) (+ 1 n)) ) )
  (length-iter s 0) )
```

Recursive call is a tail call

A stream is a Scheme list (linked list), but the rest of the list is computed on demand.

The rest of a stream is a promise. When you force a promise, you force evaluation of the expression

```
scm> (define s (cons-stream 1 (cons-stream 2 nil)))
s
scm> s
(1 . #[promise (not forced)])
scm> (car s)
1
scm> (cdr s)
#[promise (not forced)]
scm> (stream-cdr s)
(2 . #[promise (not forced)])
scm> s
(1 . #[promise (forced)])
```



You can explicitly create promises by using the delay special form. To force a

```
scm> (define x (/ 1 0))
ZeroDivisionError
scm> (define y (delay (/ 1 0)))
y
scm> y
#[promise (not forced)]
scm> (force y)
ZeroDivisionError
```

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

Infinite stream of integers starting at first

```
(define (integers first)
    (cons-stream first
                 (integers (+ first 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
```

A **simple fact** in Logic declares a relation to be true.

```
logic> (fact (father vader luke))
logic> (fact (father vader leia))
```

You can make **queries** in Logic:

```
logic> (query (parent vader luke))
Success!
logic> (query (parent luke vader))
Failed.
logic> (query (parent ?who luke))
Success!
who: vader
logic> (query (parent vader ?who))
who: luke
who: leia
```

Variables start with a question mark (?)

Logic will figure out all symbols that can fit the variable

A **compound fact** consists of a **conclusion** and one or more **hypotheses**.

```
(fact (grandparent ?x ?y)
      (parent ?x ?z)
      (parent ?z ?y))
```

?x is a grandparent of ?y if
- **?x is the parent of some ?z**, AND
- **?z is the parent of ?y**

A **recursive fact** is a compound fact where one or more of the hypotheses are recursive.

```
(fact (ancestor ?x ?y)
      (parent ?x ?z))
```

Base case

```
(fact (ancestor ?x ?y)
      (parent ?x ?z)
      (ancestor ?z ?y))
```

Recursive case

?x is an ancestor of ?y if
- **?x is the parent of some ?z**
- **?z is an ancestor of ?y**

Dot notation splits a list into first and rest

```
(fact (in ?elem (?elem . ?rest)))
(fact (in ?elem (?first . ?rest))
      (in ?elem ?rest))
```

An element is in a list if
- the element is the first element of the list, OR
- the element is in the rest of the list

```
logic> (in 4 (4 3 2 1))
Success!
logic> (in 4 (1 2 3 4))
Success!
logic> (in ?x (1 2 3 4))
Success!
x: 1
x: 2
x: 3
x: 4
```

```
logic> (in 4 (1 2 3 4))
Success!

Bindings:
elem: 4
first: 1
rest: (2 3 4)
```

A **table** has columns and rows

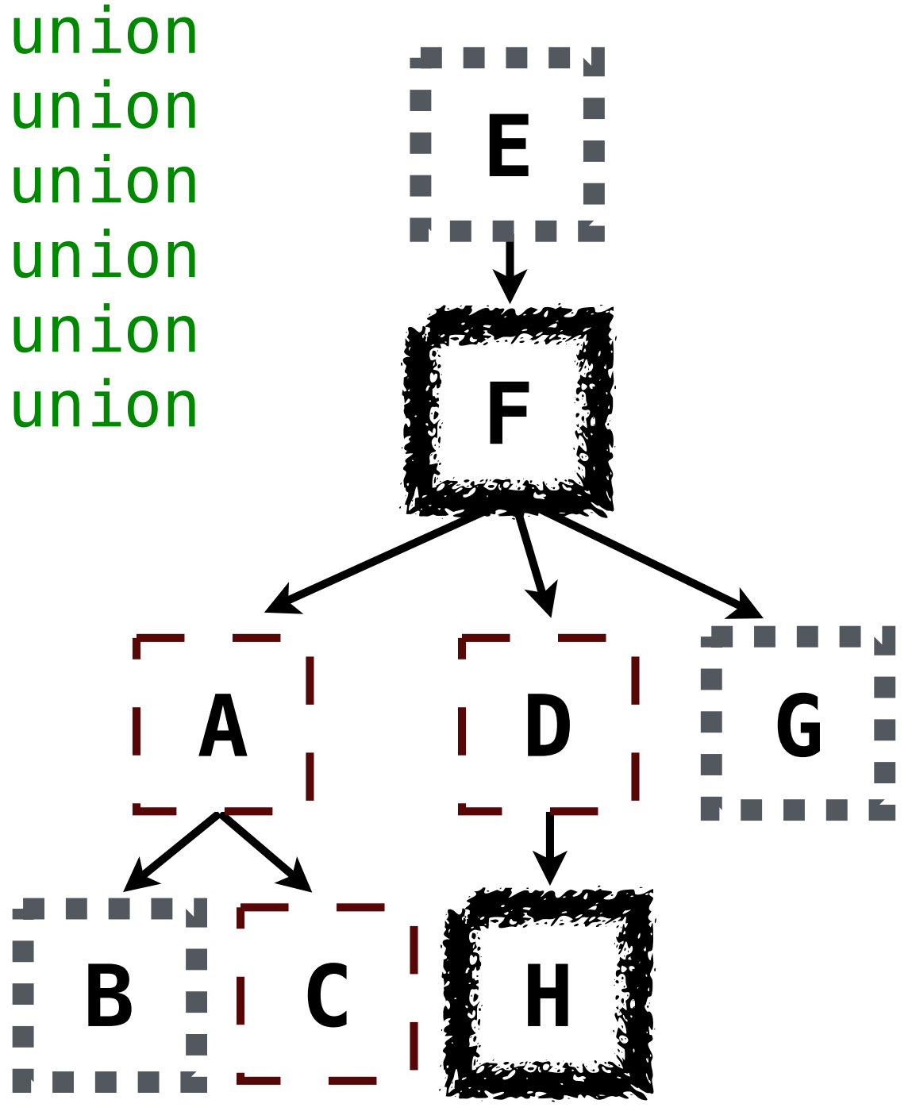
A **column** has a name and a type

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

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

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

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

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

weight/legs	count(*)
5	2
2	2