Scheme

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)
```

Scheme Lists

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)
```

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

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)
```

Scheme Special Forms

```
(define size 5) ; =>  size
(if (> size 0) size (- size)) ; =>  5
(cond ((> size 0) size) ((= size 0) 0) (else (- size))) ; =>  5
(and (> size 1) (< size 10)) ; =>  #t
(or (> size 1) (< size 3)) ; =>  #t
(let ((a size) (b (+ 1 2))) (* 2 a b)) ; =>  30
(begin (define x (+ size 1)) (* x 2)) ; =>  12
((lambda (x y) (+ x y size)) size (+ 1 2)) ; =>  13
(define (add-two x y) (+ x y)) ; =>  add-two
`(+ size (- ,size) ,(* 3 4)) ; =>  (+ size (- 5) 12)
```

Scheme Built-In Procedures

```
(+ 2 5 1) ; =>  8                  (null? '(1 2)) ; =>  #f
(- 9) ; =>  -9                     (null? '()) ; =>  #t
(- 9 3 2) ; =>  4                  (= 1 2) ; =>  #f
(* 2 5) ; =>  10                   (>= 2 1) ; =>  #t
(/ 7 2) ; =>  3.5                  (even? 2) ; =>  #t
(/ 16 2 2 2) ; =>  2               (equal? '(1 2) '(1 2)) ; => #t
(abs -1) ; =>  1                   (eq? '(1 2) '(1 2)) ; => #f
(remainder 7 3) ; =>  1            (not (> 1 2)) ; =>  #t

(append '(1 2) '(3 4)) ; =>  (1 2 3 4)
(length '(1 2 3 4)) ; =>  4
(map (lambda (x) (+ x size)) '(2 3 4)) ; =>  (7 8 9)
(filter odd? '(2 3 4)) ; =>  (3)
(reduce + '(1 2 3 4 5)) ; =>  15
(list 1 2 3 4) ; =>  (1 2 3 4)
(list (cons 1 nil) size 'size) ; =>  ((1) 5 size)
(list (or #f #t) (or) (or 1 2)) ; =>  (#t #f 1)
(list (and #f #t) (and) (and 1 2)) ; =>  (#f #t 2)
(eval + '(* 5 (* 4 (* 3 (* 2 (* 1 1)))))) ; => 6
(apply + '(1 2 3)) ; => 6
```

Scheme Scopes

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 as Data

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))
```

```
; Example: Making a procedure to generate a sum-while loop expression
; Sum the squares of even numbers less than 10, starting with 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
; 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)
  `(begin
     (define (f x total)
       (if ,while-condition
         (f ,update-x (+ total ,add-to-total))
         total))
     (f ,starting-x 0)))

(eval (sum-while 2 '(< x 10) '(* x x) '(+ x 2))) ; => 120
(eval (sum-while 1 '(< (* x x) 50) 'x '(+ x 1))) ; => 28
```

A function that can apply any function expression to any list of arguments:

```
(define (call-func func-expression func-args)
    (apply (eval func-expression) func-args)
)
(call-func '(lambda (a b) (+ a b)) '(3 4))  ; => 7
```

Scheme Tail Calls

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 final sub-expressions in a tail context **cond**
- The last sub-expression in a tail context **and**, **or**, **begin**, or **let**

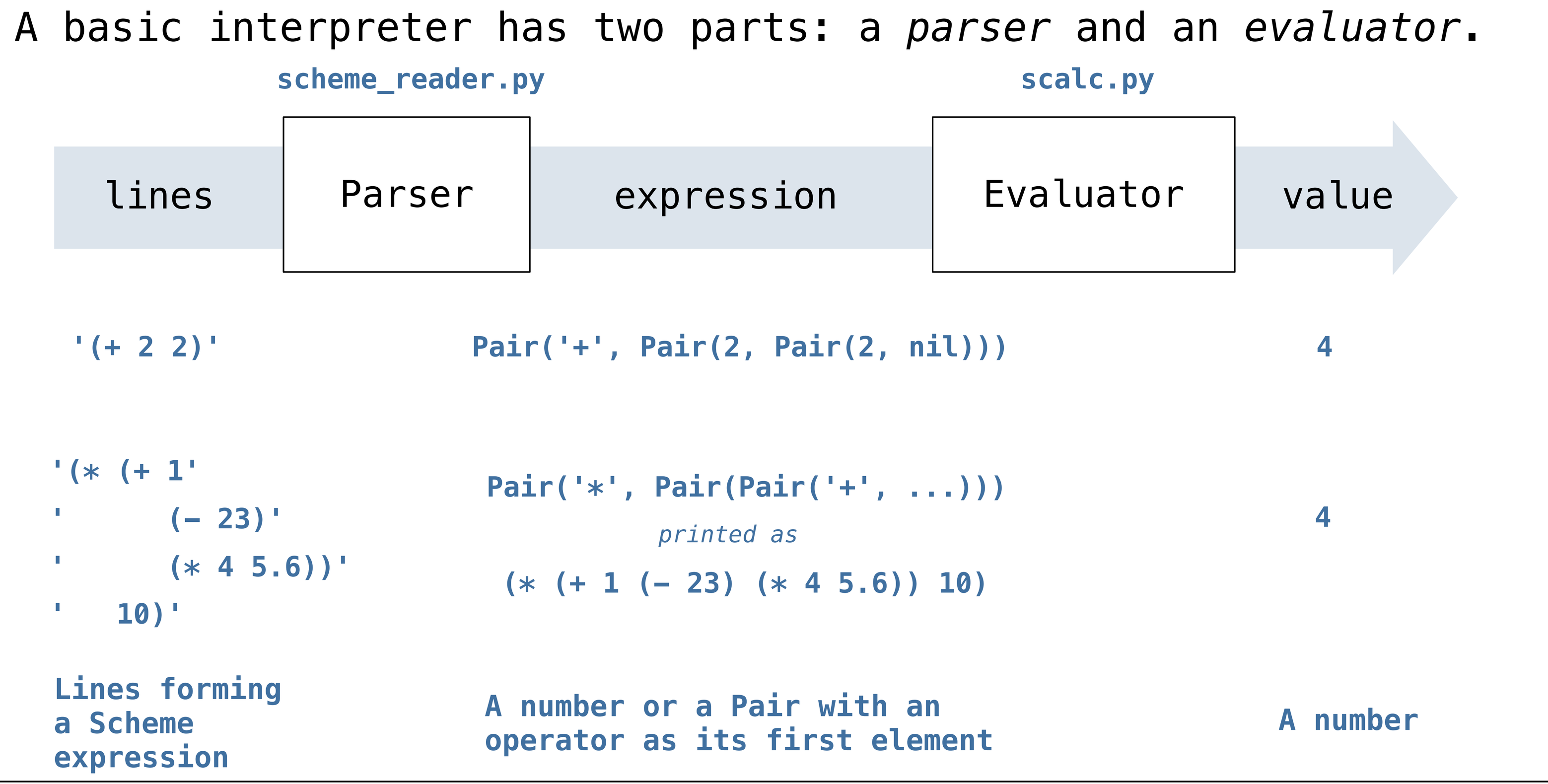
```
(define (factorial n k)               (define (length s)
  (if (= n 0) k                         (if (null? s) 0
    (factorial (- n 1)                    (+ 1 (length (cdr s))) ) ) )
               (* k n)) ) )
```

Not a tail call

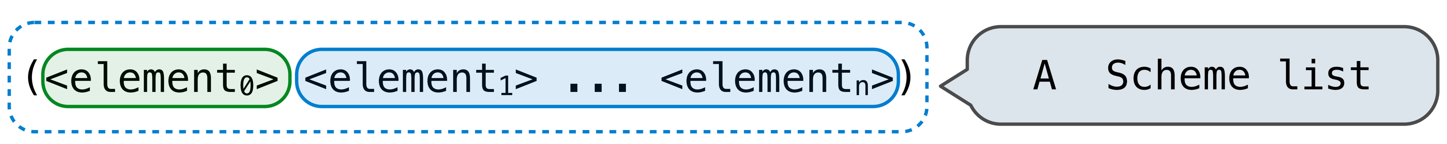
```
(define (length-tail s)
  (define (length-iter s n)
    (if (null? s) n
      (length-iter (cdr s) (+ 1 n)) ) )
  (length-iter s 0) )
```

Recursive call is a tail call

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



A Scheme list is written as elements in parentheses:



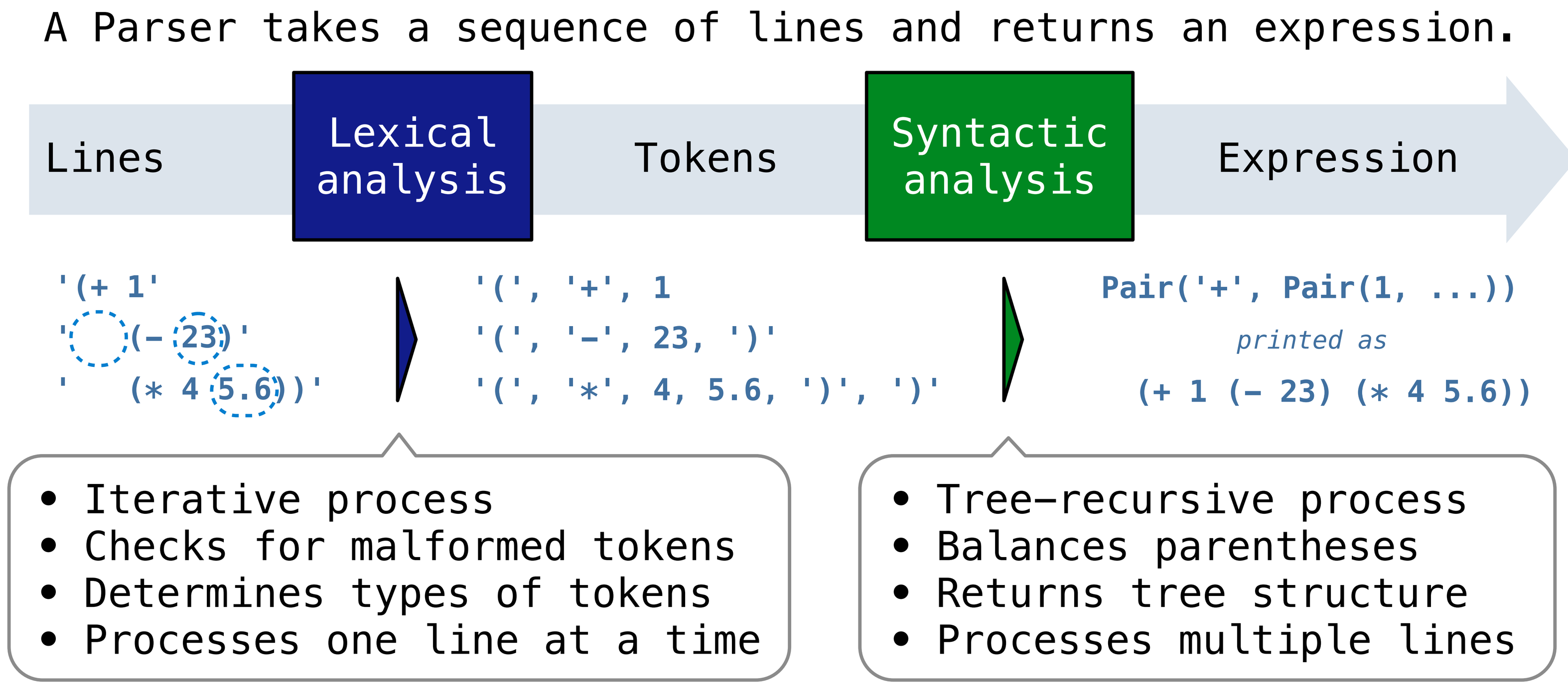
Each <element> can be a combination or atom (primitive).

(+ (* 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.

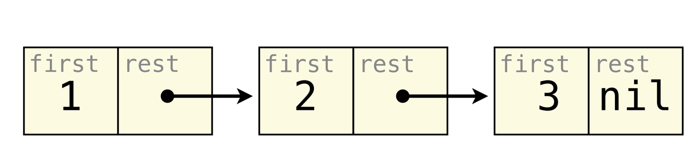


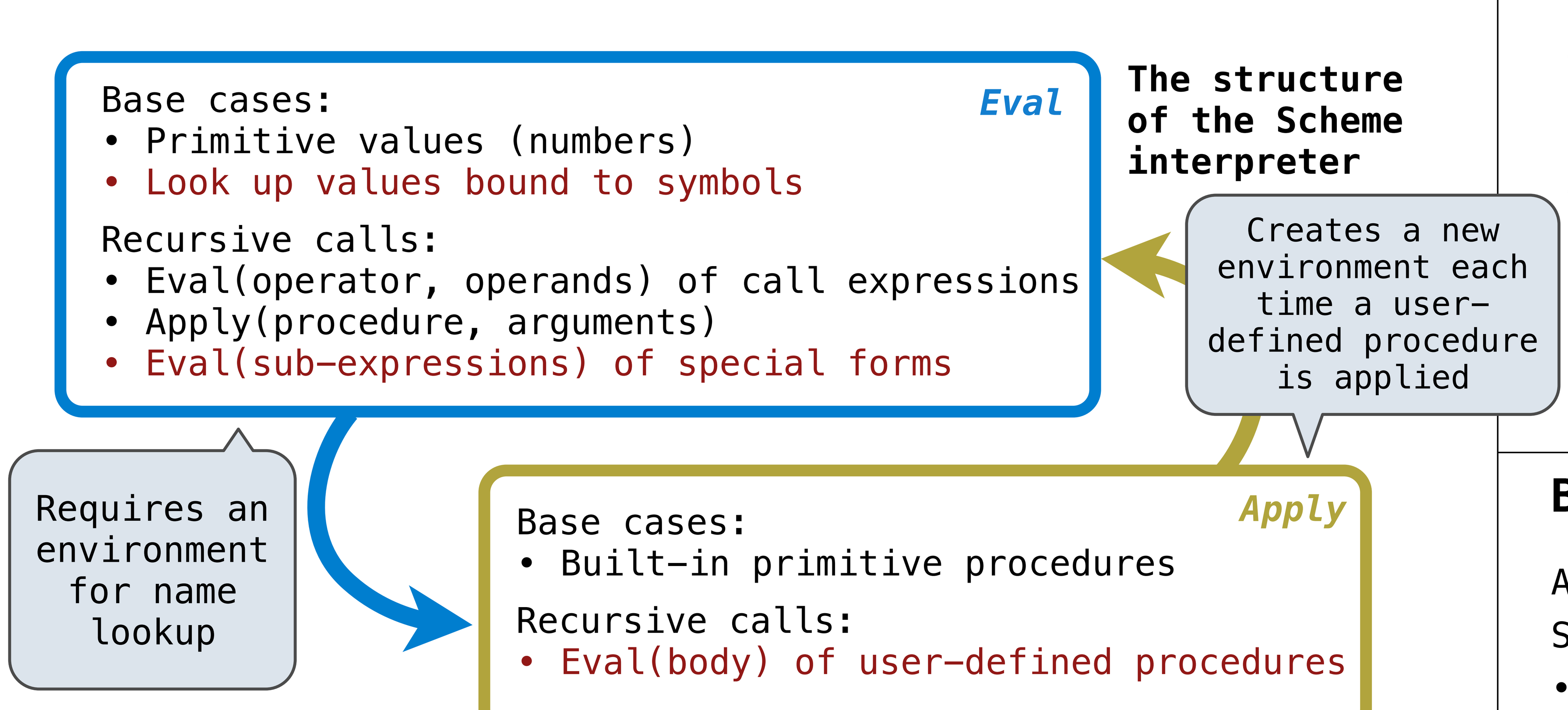
Syntactic analysis identifies the hierarchical structure of an expression, which may be nested.

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

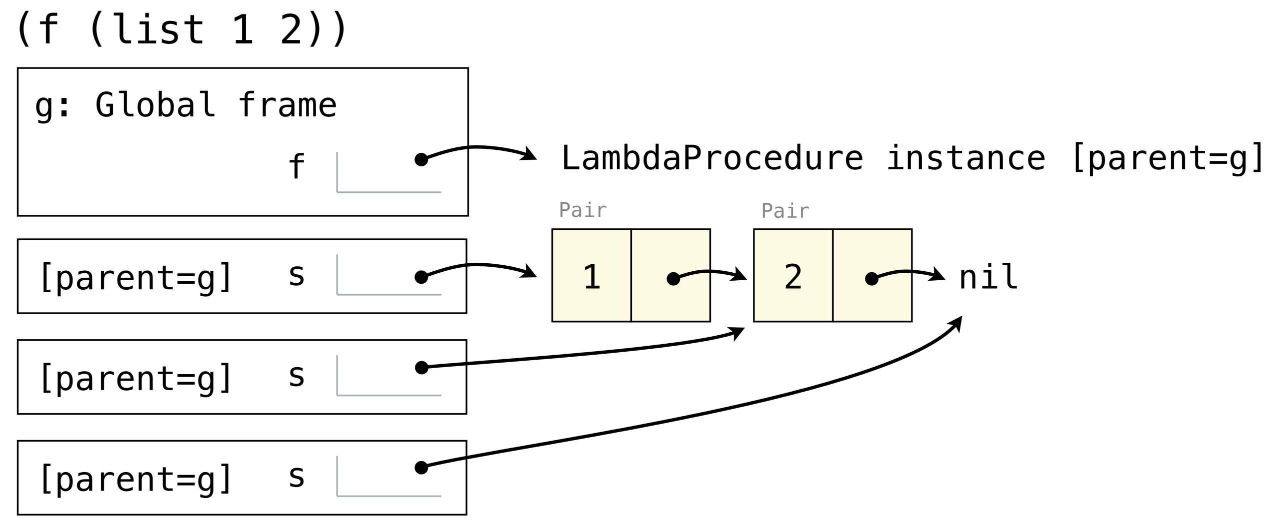
```
class Pair:
    """A pair has two instance attributes:
       first and rest.

       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)
```





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))
```

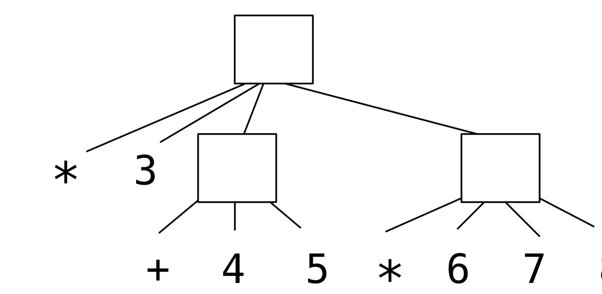


The Calculator language has primitive expressions and call expressions

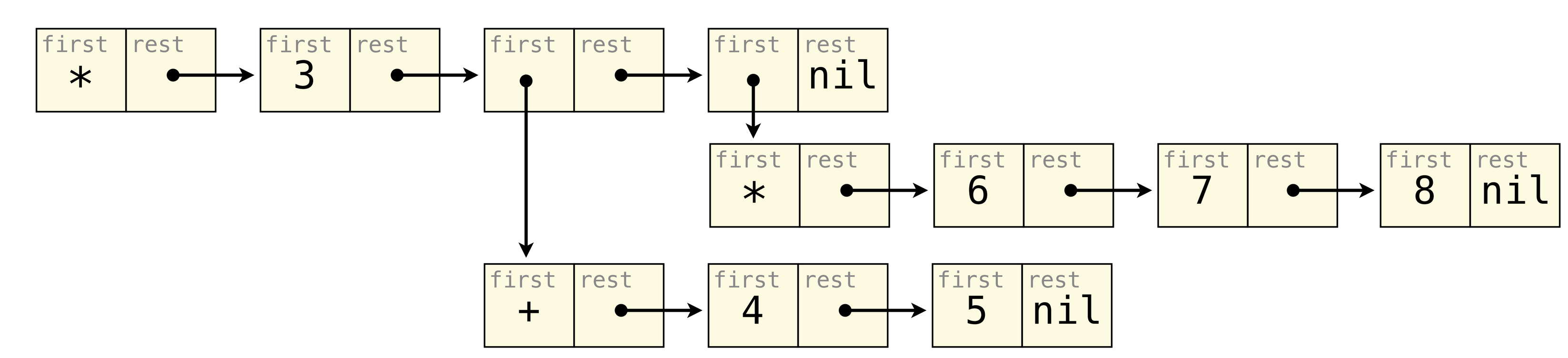
Calculator Expression

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

Expression Tree



Representation as Pairs



Exceptions in Python

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

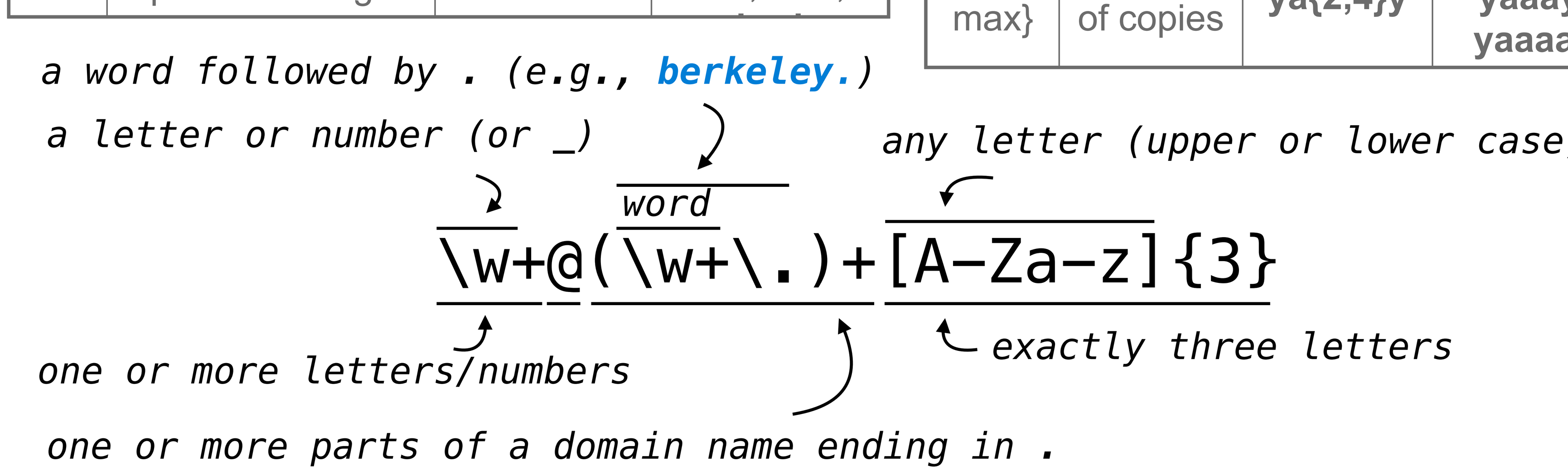
```
>>> try:
        x = 1/0
    except ZeroDivisionError as e:
        print('handling a', type(e))
        x = 0

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.

Regular Expressions

.	Matches any character	.a.	cal, ha!, (a)
\w	Matches letters, numbers or _	\wa\w	cal, dad, 3am
\d	Matches a digit	\d\d	61, 00
\s	Matches a whitespace	\d\s\d	1 2
[...]	Encloses a list of options or ranges	b[aeiou]d	bad, bed, bid, bod,

+	One or more copies	aw+	awwwww
*	Zero or more copies	b[a-z]*y	by, buy, buoy, berkeley
?	Zero or one copy	:[-o]?\)	:) :o) :-)
{min, max}	A range of copies	ya{2,4}y	yaay, yaaay, yaaaay



The | character matches either of two sequences

(Fall|Spring) 20(\d\d) matches either **Fall 2021** or **Spring 2021**

A whole group can be repeated multiple times

l(ol)+ matches **lol** and **lolol** and **lololol** but not **lolo**

The ^ and $ anchors correspond to the start and end of the full string

The \b anchor corresponds to the beginning or end of a word

Backus-Naur Form

A special symbol ?start corresponds to a complete expression.

Symbols in all caps are called terminals:

- Can only contain /regular expressions/, "text", and other TERMINALS
- No recursion is allowed within terminals

?start: numbers
numbers: INTEGER | numbers "," INTEGER
INTEGER: "0" | /-?[1-9]\d*/

- (item item ..) – Group items
- [item item ..] – Maybe. Same as (item item ..)?
- item? – Zero or one instances of item ("maybe")
- item* – Zero or more instances of item
- item+ – One or more instances of item
- item ~ n – Exactly *n* instances of item
- item ~ n..m – Between *n* to *m* instances of item