To print higher-resolution math symbols, click the
Hi-Res Fonts for Printing button on the jsMath control panel.

Chapter 3: The Structure and Interpretation of Computer Programs

Contents

3.1 Introduction

Chapters 1 and 2 describe the close connection between two fundamental elements of programming: functions and data. We saw how functions can be manipulated as data using higher-order functions. We also saw how data can be endowed with behavior using message passing and an object system. We have also studied techniques for organizing large programs, such as functional abstraction, data abstraction, class inheritance, and generic functions. These core concepts constitute a strong foundation upon which to build modular, maintainable, and extensible programs.

This chapter focuses on the third fundamental element of programming: programs themselves. A Python program is just a collection of text. Only through the process of interpretation do we perform any meaningful computation based on that text. A programming language like Python is useful because we can define an *interpreter*, a program that carries out Python's evaluation and execution procedures. It is no exaggeration to regard this as the most fundamental idea in programming, that an interpreter, which determines the meaning of expressions in a programming language, is just another program.

To appreciate this point is to change our images of ourselves as programmers. We come to see ourselves as designers of languages, rather than only users of languages designed by others.

3.1.1 Programming Languages

In fact, we can regard many programs as interpreters for some language. For example, the constraint propagator from the previous chapter has its own primitives and means of combination. The constraint language was quite specialized: it provided a declarative method for describing a certain class of mathematical relations, not a fully general language for describing computation. As another example, the object system implemented in the previous chapter created a new language for expressing class and inheritance relationships. While we have been designing languages of a sort already, the material of this chapter will greatly expand the range of languages we can interpret.

Programming languages vary widely in their syntactic structures, features, and domain of application. Among general purpose programming languages, the constructs of function definition and function application are pervasive. On the other hand, powerful languages exist that do not include an object system, higher-order functions, assignment, or even control constructs like `while` and `for` statements. To illustrate just how different languages can be, we will introduce Scheme as an example of a powerful and expressive programming language that includes few built-in features. The subset of Scheme introduced here does not allow mutable values at all.

In this chapter, we study the design of interpreters and the computational processes that they create when executing programs. The prospect of designing an interpreter for a general programming language may seem daunting. After all, interpreters are programs that can carry out any possible computation, depending on their input. However, many interpreters have an elegant common structure: two mutually recursive functions. The first evaluates expressions in environments; the second applies functions to arguments.

These functions are *recursive* in that they are defined in terms of each other: applying a function requires evaluating the expressions in its body, while evaluating an expression may involve applying one or more functions. We covered recursive functions in Chapter 1, and in the next section of this chapter, we will

focus on the evolution of recursive processes. We will then turn to recursive data structures, which will prove essential to understanding the design of an interpreter. The end of the chapter focuses on three new languages and the task of implementing interpreters for them.

3.2 Functions and the Processes They Generate

A function is a pattern for the *local evolution* of a computational process. It specifies how each stage of the process is built upon the previous stage. We would like to be able to make statements about the overall behavior of a process whose local evolution has been specified by one or more functions. This analysis is very difficult to do in general, but we can at least try to describe some typical patterns of process evolution.

In Chapter 1, we saw some common "shapes" for processes generated by simple functions, such as linear and tree recursion. In this section, we will investigate the rates at which these processes consume the important computational resources of time and space.

3.2.1 Memoization

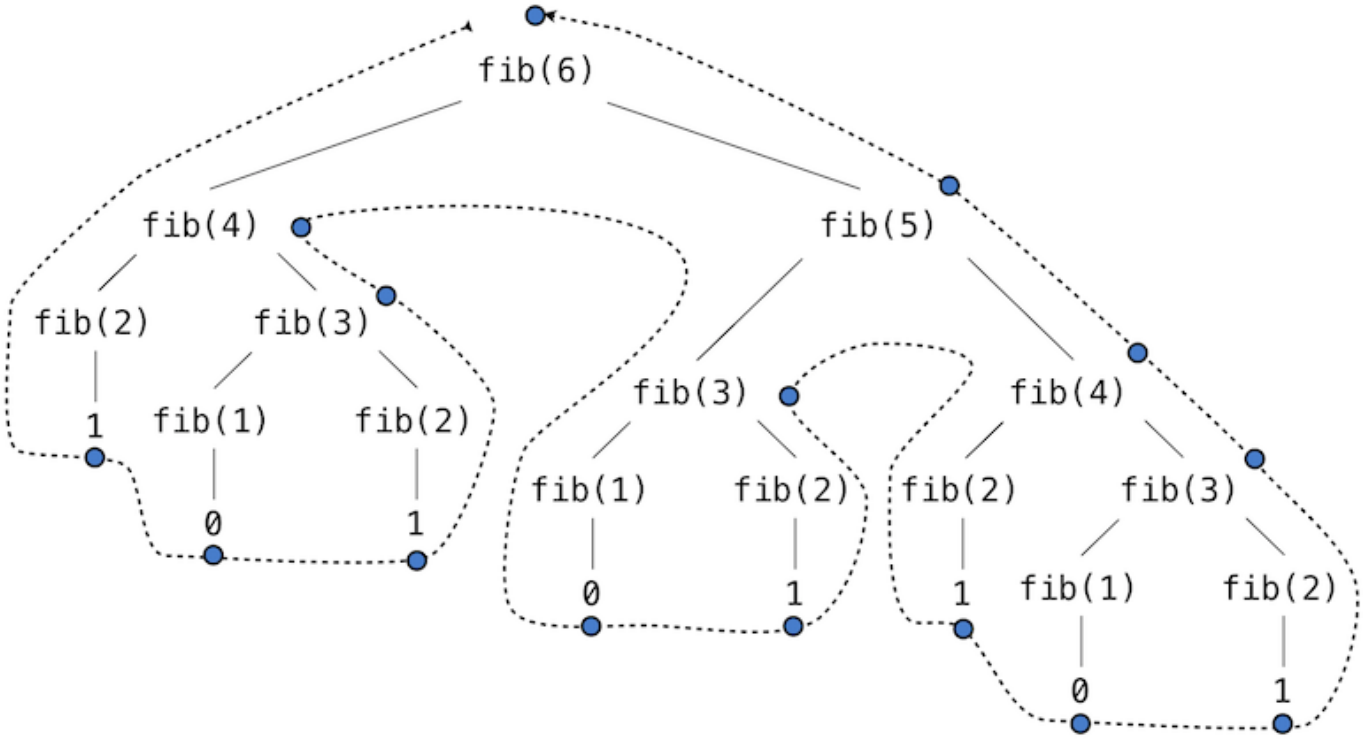
In Chapter 1, we saw multiple implementations of a function to compute Fibonacci numbers. The recursive version was as follows:

```
1  def fib(n):
2      if n == 1:
3          return 0
4      if n == 2:
5          return 1
6      return fib(n-2) + fib(n-1)
7
8  result = fib(6)
```

Edit code

< Back Step 1 of 59 Forward >

The recursive definition is tremendously appealing, since it exactly mirrors the familiar definition of Fibonacci numbers. However, consider the pattern of computation that results from evaluating `fib(6)`, shown below. To compute `fib(6)`, we compute `fib(5)` and `fib(4)`. To compute `fib(5)`, we compute `fib(4)` and `fib(3)`.



This recursive implementation is a terribly inefficient way to compute Fibonacci numbers because it does so much redundant computation. Notice that the entire computation of `fib(4)` -- almost half the work -- is duplicated. In fact, it is not hard to show that the number of times the function will compute `fib(1)` or `fib(2)` (the number of leaves in the tree, in general) is precisely `fib(n+1)`. To get an idea of how bad this is, one can show that the value of `fib(n)` grows exponentially with `n`. `fib(40)` is 63,245,986! The function above uses a number of steps that grows exponentially with the input.

We have also seen an iterative implementation of Fibonacci numbers, repeated here for convenience.

```
>>> def fib_iter(n):
        prev, curr = 1, 0  # curr is the first Fibonacci number.
        for _ in range(n-1):
            prev, curr = curr, prev + curr
        return curr
```

The state that we must maintain in this case consists of the current and previous Fibonacci numbers. Implicitly, the `for` statement also keeps track of the iteration count. This definition does not reflect the standard mathematical definition of Fibonacci numbers as clearly as the recursive approach. However, the amount of computation required in the iterative implementation is only linear in `n`, rather than exponential. Even for small values of `n`, this difference can be enormous.

One should not conclude from this difference that tree-recursive processes are useless. When we consider processes that operate on hierarchically structured data rather than numbers, we will find that tree recursion is a natural and powerful tool.

Furthermore, tree-recursive processes can often be made more efficient through *memoization*, a powerful technique for increasing the efficiency of recursive functions that repeat computation. A memoized function

will store the return value for any arguments it has previously received. A second call to `fib(4)` would not evolve the same complex process as the first, but instead would immediately return the stored result computed by the first call. If the memoized function is a pure function, then memoization is guaranteed not to change the result.

Memoization can be expressed naturally as a higher-order function, which can also be used as a decorator. The definition below creates a *cache* of previously computed results, indexed by the arguments from which they were computed. The use of a dictionary will require that the argument to the memoized function be immutable.

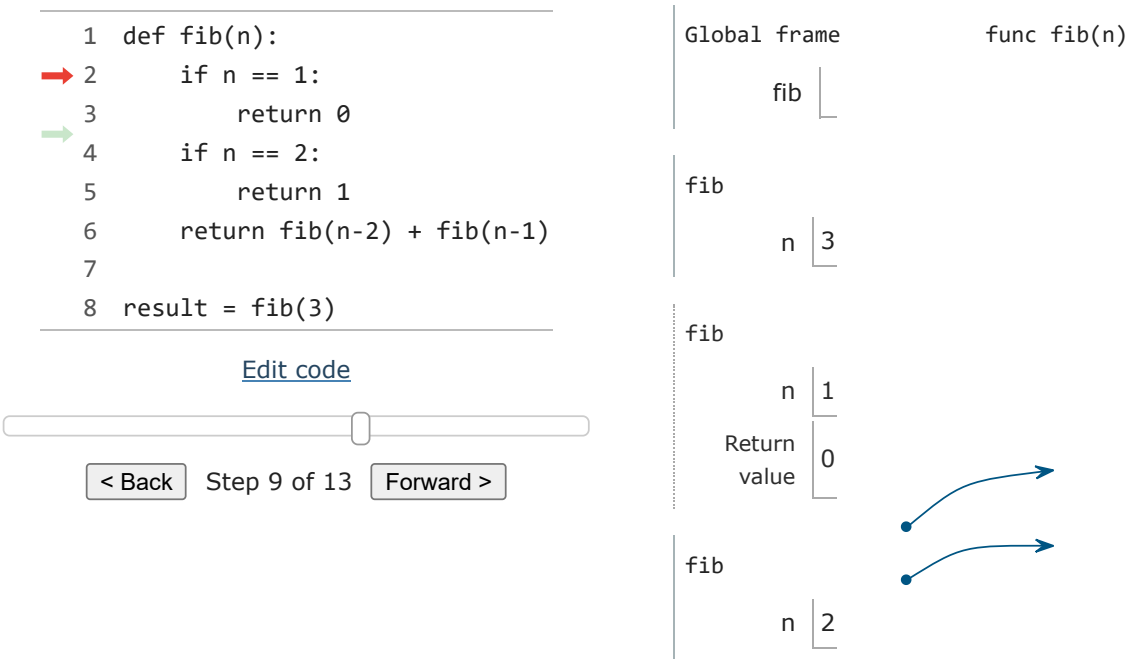
```
>>> def memo(f):
        """Return a memoized version of single-argument function f."""
        cache = {}
        def memoized(n):
            if n not in cache:
                cache[n] = f(n)
            return cache[n]
        return memoized

>>> @memo
    def fib(n):
        if n == 1:
            return 0
        if n == 2:
            return 1
        return fib(n-2) + fib(n-1)

>>> fib(40)
63245986
```

The amount of computation time saved by memoization in this case is substantial. The memoized, recursive *fib* function and the iterative *fib_iter* function both require an amount of time to compute that is only a linear function of their input `n`. To compute `fib(40)`, the body of `fib` is executed 40 times, rather than 102,334,155 times in the unmemoized recursive case.

Space. To understand the space requirements of a function, we must specify generally how memory is used, preserved, and reclaimed in our environment model of computation. In evaluating an expression, we must preserve all *active* environments and all values and frames referenced by those environments. An environment is active if it provides the evaluation context for some expression being evaluated.

For example, when evaluating `fib`, the interpreter proceeds to compute each value in the order shown previously, traversing the structure of the tree. To do so, it only needs to keep track of those nodes that are above the current node in the tree at any point in the computation. The memory used to evaluate the rest of the branches can be reclaimed because it cannot affect future computation. In general, the space required for tree-recursive functions will be proportional to the maximum depth of the tree.

The diagram below depicts the environment created by evaluating `fib(3)`. In the process of evaluating the return expression for the initial application of `fib`, the expression `fib(n-2)` is evaluated, yielding a value of `0`. Once this value is computed, the corresponding environment frame (grayed out) is no longer needed: it is not part of an active environment. Thus, a well-designed interpreter can reclaim the memory that was used to store this frame. On the other hand, if the interpreter is currently evaluating `fib(n-1)`, then the environment created by this application of `fib` (in which `n` is `2`) is active. In turn, the environment originally created to apply `fib` to `3` is active because its return value has not yet been computed.

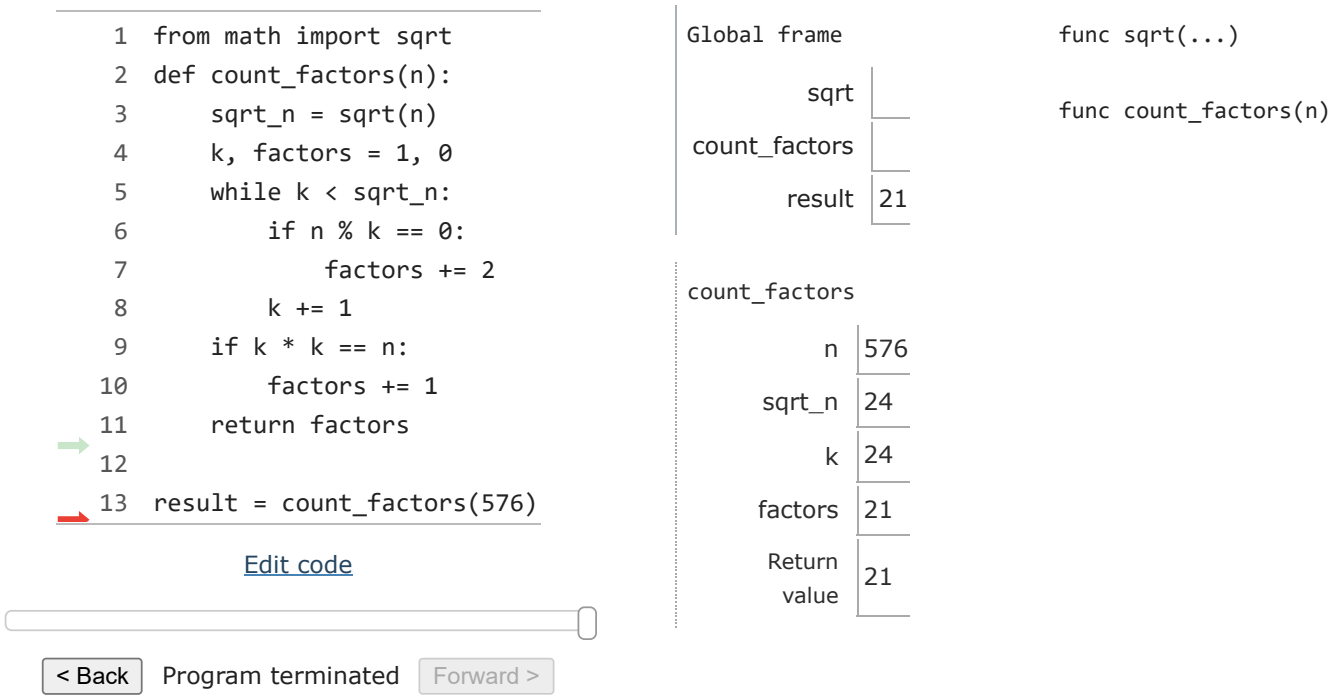


In the case of `memo`, the environment associated with the function it returns (which contains `cache`) must be preserved as long as some name is bound to that function in an active environment. The number of entries in the `cache` dictionary grows linearly with the number of unique arguments passed to `fib`, which scales linearly with the input. On the other hand, the iterative implementation requires only two numbers to be tracked during computation: `prev` and `curr`, giving it a constant size.

Memoization exemplifies a common pattern in programming that computation time can often be decreased at the expense of increased use of space, or vis versa.

3.2.2 Orders of Growth

The previous examples illustrate that processes can differ considerably in the rates at which they consume the computational resources of space and time. For some functions, we can exactly predict the number of steps in the computational process evolved by those functions. For example, consider the function `count_factors` below that counts the number of integers that evenly divide an input `n`, by attempting to divide it by every integer less than or equal to its square root. The implementation takes advantage of the fact that if k divides n and $k < \sqrt{n}$, then there is another factor $j = n/k$ such that $j > \sqrt{n}$.



The total number of times this process executes the body of the `while` statement is the greatest integer less than $\sqrt{n}$. Hence, we can say that the amount of time used by this function, typically denoted $R(n)$, scales with the square root of the input, which we write as $R(n) = \sqrt{n}$.

For most functions, we cannot exactly determine the number of steps or iterations they will require. One convenient way to describe this difference is to use the notion of *order of growth* to obtain a coarse measure of the resources required by a process as the inputs become larger.

Let n be a parameter that measures the size of the problem to be solved, and let $R(n)$ be the amount of resources the process requires for a problem of size n. In our previous examples we took n to be the number for which a given function is to be computed, but there are other possibilities. For instance, if our goal is to compute an approximation to the square root of a number, we might take n to be the number of digits of accuracy required. In general there are a number of properties of the problem with respect to which it will be desirable to analyze a given process. Similarly, $R(n)$ might measure the amount of memory used, the number of elementary machine operations performed, and so on. In computers that do only a fixed number of operations at a time, the time required to evaluate an expression will be proportional to the number of elementary machine operations performed in the process of evaluation.

We say that $R(n)$ has order of growth $\Theta(f(n))$, written $R(n) = \Theta(f(n))$ (pronounced "theta of $f(n)$"), if there are positive constants k_1 and k_2 independent of n such that

$$k_1 \cdot f(n) \leq R(n) \leq k_2 \cdot f(n)$$

for any sufficiently large value of n. In other words, for large n, the value $R(n)$ is sandwiched between two values that both scale with $f(n)$:

- A lower bound $k_1 \cdot f(n)$ and

- An upper bound $k_2 \cdot f(n)$

For instance, the number of steps to compute $n!$ grows proportionally to the input n. Thus, the steps required for this process grows as $\Theta(n)$. We also saw that the space required for the recursive implementation `fact` grows as $\Theta(n)$. By contrast, the iterative implementation `fact_iter` takes a similar number of steps, but the space it requires stays constant. In this case, we say that the space grows as $\Theta(1)$.

The number of steps in our tree-recursive Fibonacci computation `fib` grows exponentially in its input n. In particular, one can show that the nth Fibonacci number is the closest integer to

$$\frac{\phi^{n-2}}{\sqrt{5}}$$

where ϕ is the golden ratio:

$$\phi = \frac{1+\sqrt{5}}{2} \approx 1.6180$$

We also stated that the number of steps scales with the resulting value, and so the tree-recursive process requires $\Theta(\phi^n)$ steps, a function that grows exponentially with n.

Orders of growth provide only a crude description of the behavior of a process. For example, a process requiring n^2 steps and a process requiring $1000 \cdot n^2$ steps and a process requiring $3 \cdot n^2 + 10 \cdot n + 17$ steps all have $\Theta(n^2)$ order of growth. There are certainly cases in which an order of growth analysis is too coarse a method for deciding between two possible implementations of a function.

However, order of growth provides a useful indication of how we may expect the behavior of the process to change as we change the size of the problem. For a $\Theta(n)$ (linear) process, doubling the size will roughly double the amount of resources used. For an exponential process, each increment in problem size will multiply the resource utilization by a constant factor. The next example examines an algorithm whose order of growth is logarithmic, so that doubling the problem size increases the resource requirement by only a constant amount.

3.2.3 Example: Exponentiation

Consider the problem of computing the exponential of a given number. We would like a function that takes as arguments a base `b` and a positive integer exponent `n` and computes b^n. One way to do this is via the recursive definition

$$b^n = b \cdot b^{n-1}$$
$$b^0 = 1$$

which translates readily into the recursive function

```
>>> def exp(b, n):
        if n == 0:
            return 1
        return b * exp(b, n-1)
```

This is a linear recursive process that requires $\Theta(n)$ steps and $\Theta(n)$ space. Just as with factorial, we can readily formulate an equivalent linear iteration that requires a similar number of steps but constant space.

```
>>> def exp_iter(b, n):
        result = 1
        for _ in range(n):
            result = result * b
        return result
```

We can compute exponentials in fewer steps by using successive squaring. For instance, rather than computing b^8 as

$$b \cdot (b \cdot (b \cdot (b \cdot (b \cdot (b \cdot (b \cdot b))))))$$

we can compute it using three multiplications:

$$\begin{aligned} b^2 &= b \cdot b \\ b^4 &= b^2 \cdot b^2 \\ b^8 &= b^4 \cdot b^4 \end{aligned}$$

This method works fine for exponents that are powers of 2. We can also take advantage of successive squaring in computing exponentials in general if we use the recursive rule

$$b^n = \begin{cases} (b^{\frac{1}{2}n})^2 & \text{if } n \text{ is even} \\ b \cdot b^{n-1} & \text{if } n \text{ is odd} \end{cases}$$

We can express this method as a recursive function as well:

```
>>> def square(x):
        return x*x

>>> def fast_exp(b, n):
        if n == 0:
            return 1
        if n % 2 == 0:
            return square(fast_exp(b, n//2))
        else:
            return b * fast_exp(b, n-1)

>>> fast_exp(2, 100)
1267650600228229401496703205376
```

The process evolved by `fast_exp` grows logarithmically with `n` in both space and number of steps. To see this, observe that computing b^{2n} using `fast_exp` requires only one more multiplication than computing b^n. The size of the exponent we can compute therefore doubles (approximately) with every new multiplication we are allowed. Thus, the number of multiplications required for an exponent of `n` grows about as fast as the logarithm of `n` base `2`. The process has $\Theta(\log n)$ growth. The difference between $\Theta(\log n)$ growth and $\Theta(n)$ growth becomes striking as n becomes large. For example, `fast_exp` for `n` of `1000` requires only `14` multiplications instead of `1000`.

3.3 Recursive Data Structures

In Chapter 2, we introduced the notion of a pair as a primitive mechanism for glueing together two objects into one. We showed that a pair can be implemented using a built-in tuple. The *closure* property of pairs indicated that either element of a pair could itself be a pair.

This closure property allowed us to implement the recursive list data abstraction, which served as our first type of sequence. Recursive lists are most naturally manipulated using recursive functions, as their name and structure would suggest. In this section, we discuss functions for creating and manipulating recursive lists and other recursive data structures.

3.3.1 Processing Recursive Lists

Recall that the recursive list abstract data type represented a list as a first element and the rest of the list. We previously implemented recursive lists using functions, but at this point we can re-implement them using a class. Below, the length (`__len__`) and element selection (`__getitem__`) functions are written recursively to demonstrate typical patterns for processing recursive lists.

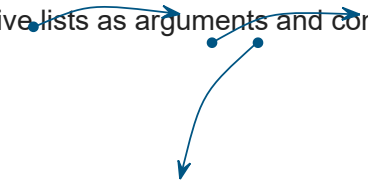
```
>>> class Rlist(object):
        """A recursive list consisting of a first element and the rest."""
        class EmptyList(object):
            def __len__(self):
                return 0
        empty = EmptyList()
        def __init__(self, first, rest=empty):
            self.first = first
            self.rest = rest
        def __repr__(self):
            args = repr(self.first)
            if self.rest is not Rlist.empty:
                args += ', {0}'.format(repr(self.rest))
            return 'Rlist({0})'.format(args)
        def __len__(self):
            return 1 + len(self.rest)
        def __getitem__(self, i):
            if i == 0:
```

```
            return self.first
        return self.rest[i-1]
```

The definitions of `__len__` and `__getitem__` are in fact recursive, although not explicitly so. The built-in Python function `len` looks for a method called `__len__` when applied to a user-defined object argument. Likewise, the subscript operator looks for a method called `__getitem__`. Thus, these definitions will end up calling themselves. Recursive calls on the rest of the list are a ubiquitous pattern in recursive list processing. This class definition of a recursive list interacts properly with Python's built-in sequence and printing operations.

```
>>> s = Rlist(1, Rlist(2, Rlist(3)))
>>> s.rest
Rlist(2, Rlist(3))
>>> len(s)
3
>>> s[1]
2
```

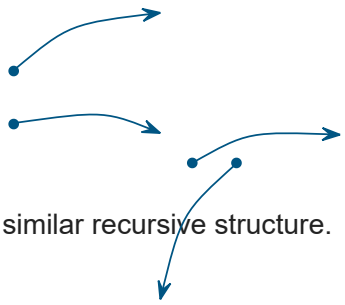
Operations that create new lists are particularly straightforward to express using recursion. For example, we can define a function `extend_rlist`, which takes two recursive lists as arguments and combines the elements of both into a new list.



```
>>> def extend_rlist(s1, s2):
        if s1 is Rlist.empty:
            return s2
        return Rlist(s1.first, extend_rlist(s1.rest, s2))

>>> extend_rlist(s.rest, s)
Rlist(2, Rlist(3, Rlist(1, Rlist(2, Rlist(3)))))
```

Likewise, mapping a function over a recursive list exhibits a similar pattern.

```
>>> def map_rlist(s, fn):
        if s is Rlist.empty:
            return s
        return Rlist(fn(s.first), map_rlist(s.rest, fn))

>>> map_rlist(s, square)
Rlist(1, Rlist(4, Rlist(9)))
```



Filtering includes an additional conditional statement, but otherwise has a similar recursive structure.

```
>>> def filter_rlist(s, fn):
        if s is Rlist.empty:
            return s
        rest = filter_rlist(s.rest, fn)
```

```
        if fn(s.first):
            return Rlist(s.first, rest)
        return rest

>>> filter_rlist(s, lambda x: x % 2 == 1)
Rlist(1, Rlist(3))
```

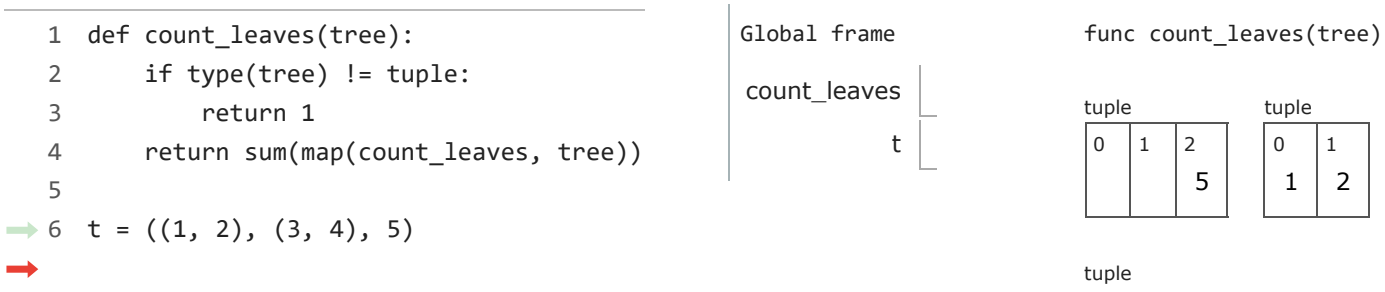
Recursive implementations of list operations do not, in general, require local assignment or `while` statements. Instead, recursive lists are taken apart and constructed incrementally as a consequence of function application. As a result, they have linear orders of growth in both the number of steps and space required.

3.3.2 Hierarchical Structures

Hierarchical structures result from the closure property of data, which asserts for example that tuples can contain other tuples. For instance, consider this nested representation of the numbers 1 through 5. This tuple is a length-three sequence, of which the first two elements are themselves tuples. A tuple that contains tuples or other values is a tree.

In a tree, each subtree is itself a tree. As a base condition, any bare element that is not a tuple is itself a simple tree, one with no branches. That is, the numbers are all trees, as is the pair `(1, 2)` and the structure as a whole.

Recursion is a natural tool for dealing with tree structures, since we can often reduce operations on trees to operations on their branches, which reduce in turn to operations on the branches of the branches, and so on, until we reach the leaves of the tree. As an example, we can implement a `count_leaves` function, which returns the total number of leaves of a tree. Step through this function to see how the leaves are counted.



```
7  result = count_leaves(t)
```

0	1
3	4

Edit code

< Back Step 3 of 27 Forward >

Just as `map` is a powerful tool for dealing with sequences, mapping and recursion together provide a powerful general form of computation for manipulating trees. For instance, we can square all leaves of a tree using a higher-order recursive function `map_tree` that is structured quite similarly to `count_leaves`.

```
>>> def map_tree(tree, fn):
        if type(tree) != tuple:
            return fn(tree)
        return tuple(map_tree(branch, fn) for branch in tree)

>>> map_tree(big_tree, square)
((((1, 4), 9, 16), ((1, 4), 9, 16)), 25)
```

Internal values. The trees described above have values only at the leaves. Another common representation of tree-structured data has values for the internal nodes of the tree as well. We can represent such trees using a class.

```
>>> class Tree(object):
        def __init__(self, entry, left=None, right=None):
            self.entry = entry
            self.left = left
            self.right = right
        def __repr__(self):
            args = repr(self.entry)
            if self.left or self.right:
                args += ', {0}, {1}'.format(repr(self.left), repr(self.right))
            return 'Tree({0})'.format(args)
```

The `Tree` class can represent, for instance, the values computed in an expression tree for the recursive implementation of `fib`, the function for computing Fibonacci numbers. The function `fib_tree(n)` below returns a `Tree` that has the nth Fibonacci number as its `entry` and a trace of all previously computed Fibonacci numbers within its branches.

```
>>> def fib_tree(n):
        """Return a Tree that represents a recursive Fibonacci calculation."""
        if n == 1:
            return Tree(0)
        if n == 2:
            return Tree(1)
        left = fib_tree(n-2)
```

```
        right = fib_tree(n-1)
        return Tree(left.entry + right.entry, left, right)

>>> fib_tree(5)
Tree(3, Tree(1, Tree(0), Tree(1)), Tree(2, Tree(1), Tree(1, Tree(0), Tree(1))))
```

This example shows that expression trees can be represented programmatically using tree-structured data. This connection between nested expressions and tree-structured data type plays a central role in our discussion of designing interpreters later in this chapter.

3.3.3 Sets

In addition to the list, tuple, and dictionary, Python has a fourth built-in container type called a `set`. Set literals follow the mathematical notation of elements enclosed in braces. Duplicate elements are removed upon construction. Sets are unordered collections, and so the printed ordering may differ from the element ordering in the set literal.

```
>>> s = {3, 2, 1, 4, 4}
>>> s
{1, 2, 3, 4}
```

Python sets support a variety of operations, including membership tests, length computation, and the standard set operations of union and intersection

```
>>> 3 in s
True
>>> len(s)
4
>>> s.union({1, 5})
{1, 2, 3, 4, 5}
>>> s.intersection({6, 5, 4, 3})
{3, 4}
```

In addition to `union` and `intersection`, Python sets support several other methods. The predicates `isdisjoint`, `issubset`, and `issuperset` provide set comparison. Sets are mutable, and can be changed one element at a time using `add`, `remove`, `discard`, and `pop`. Additional methods provide multi-element mutations, such as `clear` and `update`. The Python documentation for sets should be sufficiently intelligible at this point of the course to fill in the details.

Implementing sets. Abstractly, a set is a collection of distinct objects that supports membership testing, union, intersection, and adjunction. Adjoining an element and a set returns a new set that contains all of the original set's elements along with the new element, if it is distinct. Union and intersection return the set of elements that appear in either or both sets, respectively. As with any data abstraction, we are free to implement any functions over any representation of sets that provides this collection of behaviors.

In the remainder of this section, we consider three different methods of implementing sets that vary in their representation. We will characterize the efficiency of these different representations by analyzing the order of growth of set operations. We will use our `Rlist` and `Tree` classes from earlier in this section, which allow for simple and elegant recursive solutions for elementary set operations.

Sets as unordered sequences. One way to represent a set is as a sequence in which no element appears more than once. The empty set is represented by the empty sequence. Membership testing walks recursively through the list.

```
>>> def empty(s):
        return s is Rlist.empty

>>> def set_contains(s, v):
        """Return True if and only if set s contains v."""
        if empty(s):
            return False
        elif s.first == v:
            return True
        return set_contains(s.rest, v)

>>> s = Rlist(1, Rlist(2, Rlist(3)))
>>> set_contains(s, 2)
True
>>> set_contains(s, 5)
False
```

This implementation of `set_contains` requires $\Theta(n)$ time to test membership of an element, where n is the size of the set `s`. Using this linear-time function for membership, we can adjoin an element to a set, also in linear time.

```
>>> def adjoin_set(s, v):
        """Return a set containing all elements of s and element v."""
        if set_contains(s, v):
            return s
        return Rlist(v, s)

>>> t = adjoin_set(s, 4)
>>> t
Rlist(4, Rlist(1, Rlist(2, Rlist(3))))
```

In designing a representation, one of the issues with which we should be concerned is efficiency. Intersecting two sets `set1` and `set2` also requires membership testing, but this time each element of `set1` must be tested for membership in `set2`, leading to a quadratic order of growth in the number of steps, $\Theta(n^2)$, for two sets of size n.

```
>>> def intersect_set(set1, set2):
        """Return a set containing all elements common to set1 and set2."""
```

```
        return filter_rlist(set1, lambda v: set_contains(set2, v))

>>> intersect_set(t, map_rlist(s, square))
Rlist(4, Rlist(1))
```

When computing the union of two sets, we must be careful not to include any element twice. The `union_set` function also requires a linear number of membership tests, creating a process that also includes $\Theta(n^2)$ steps.

```
>>> def union_set(set1, set2):
        """Return a set containing all elements either in set1 or set2."""
        set1_not_set2 = filter_rlist(set1, lambda v: not set_contains(set2, v))
        return extend_rlist(set1_not_set2, set2)

>>> union_set(t, s)
Rlist(4, Rlist(1, Rlist(2, Rlist(3))))
```

Sets as ordered tuples. One way to speed up our set operations is to change the representation so that the set elements are listed in increasing order. To do this, we need some way to compare two objects so that we can say which is bigger. In Python, many different types of objects can be compared using `<` and `>` operators, but we will concentrate on numbers in this example. We will represent a set of numbers by listing its elements in increasing order.

One advantage of ordering shows up in `set_contains`: In checking for the presence of an object, we no longer have to scan the entire set. If we reach a set element that is larger than the item we are looking for, then we know that the item is not in the set:

```
>>> def set_contains(s, v):
        if empty(s) or s.first > v:
            return False
        elif s.first == v:
            return True
        return set_contains(s.rest, v)

>>> set_contains(s, 0)
False
```

How many steps does this save? In the worst case, the item we are looking for may be the largest one in the set, so the number of steps is the same as for the unordered representation. On the other hand, if we search for items of many different sizes we can expect that sometimes we will be able to stop searching at a point near the beginning of the list and that other times we will still need to examine most of the list. On average we should expect to have to examine about half of the items in the set. Thus, the average number of steps required will be about $\frac{n}{2}$. This is still $\Theta(n)$ growth, but it does save us, on average, a factor of 2 in the number of steps over the previous implementation.

We can obtain a more impressive speedup by re-implementing `intersect_set`. In the unordered representation, this operation required $\Theta(n^2)$ steps because we performed a complete scan of `set2` for each element of `set1`. But with the ordered representation, we can use a more clever method. We iterate through both sets simultaneously, tracking an element `e1` in `set1` and `e2` in `set2`. When `e1` and `e2` are equal, we include that element in the intersection.

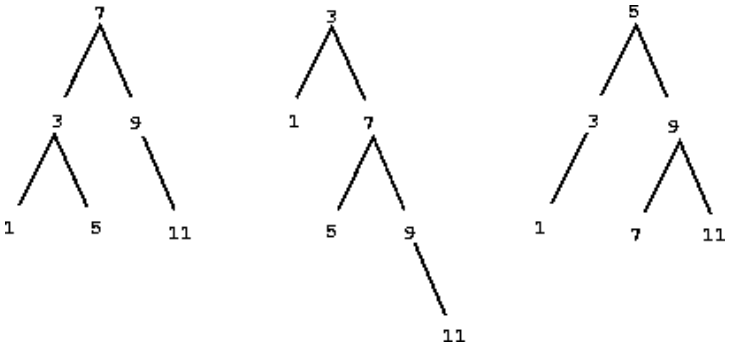
Suppose, however, that `e1` is less than `e2`. Since `e2` is smaller than the remaining elements of `set2`, we can immediately conclude that `e1` cannot appear anywhere in the remainder of `set2` and hence is not in the intersection. Thus, we no longer need to consider `e1`; we discard it and proceed to the next element of `set1`. Similar logic advances through the elements of `set2` when `e2 < e1`. Here is the function:

```
>>> def intersect_set(set1, set2):
        if empty(set1) or empty(set2):
            return Rlist.empty
        e1, e2 = set1.first, set2.first
        if e1 == e2:
            return Rlist(e1, intersect_set(set1.rest, set2.rest))
        elif e1 < e2:
            return intersect_set(set1.rest, set2)
        elif e2 < e1:
            return intersect_set(set1, set2.rest)

>>> intersect_set(s, s.rest)
Rlist(2, Rlist(3))
```

To estimate the number of steps required by this process, observe that in each step we shrink the size of at least one of the sets. Thus, the number of steps required is at most the sum of the sizes of `set1` and `set2`, rather than the product of the sizes, as with the unordered representation. This is $\Theta(n)$ growth rather than $\Theta(n^2)$ -- a considerable speedup, even for sets of moderate size. For example, the intersection of two sets of size `100` will take around `200` steps, rather than `10,000` for the unordered representation.

Adjunction and union for sets represented as ordered sequences can also be computed in linear time. These implementations are left as an exercise.

Sets as binary trees. We can do better than the ordered-list representation by arranging the set elements in the form of a tree. We use the `Tree` class introduced previously. The `entry` of the root of the tree holds one element of the set. The entries within the `left` branch include all elements smaller than the one at the root. Entries in the `right` branch include all elements greater than the one at the root. The figure below shows some trees that represent the set `{1, 3, 5, 7, 9, 11}`. The same set may be represented by a tree in a number of different ways. The only thing we require for a valid representation is that all elements in the `left` subtree be smaller than the tree `entry` and that all elements in the `right` subtree be larger.



The advantage of the tree representation is this: Suppose we want to check whether a value `v` is contained in a set. We begin by comparing `v` with `entry`. If `v` is less than this, we know that we need only search the `left` subtree; if `v` is greater, we need only search the `right` subtree. Now, if the tree is "balanced," each of these subtrees will be about half the size of the original. Thus, in one step we have reduced the problem of searching a tree of size n to searching a tree of size $\frac{n}{2}$. Since the size of the tree is halved at each step, we should expect that the number of steps needed to search a tree grows as $\Theta(\log n)$. For large sets, this will be a significant speedup over the previous representations. This `set_contains` function exploits the ordering structure of the tree-structured set.

```
>>> def set_contains(s, v):
        if s is None:
            return False
        elif s.entry == v:
            return True
        elif s.entry < v:
            return set_contains(s.right, v)
        elif s.entry > v:
            return set_contains(s.left, v)
```

Adjoining an item to a set is implemented similarly and also requires $\Theta(\log n)$ steps. To adjoin a value `v`, we compare `v` with `entry` to determine whether `v` should be added to the `right` or to the `left` branch, and having adjoined `v` to the appropriate branch we piece this newly constructed branch together with the original `entry` and the other branch. If `v` is equal to the `entry`, we just return the node. If we are asked to adjoin `v` to an empty tree, we generate a `Tree` that has `v` as the `entry` and empty `right` and `left` branches. Here is the function:

```
>>> def adjoin_set(s, v):
        if s is None:
            return Tree(v)
        if s.entry == v:
            return s
        if s.entry < v:
            return Tree(s.entry, s.left, adjoin_set(s.right, v))
        if s.entry > v:
            return Tree(s.entry, adjoin_set(s.left, v), s.right)

>>> adjoin_set(adjoin_set(adjoin_set(None, 2), 3), 1)
```

```
Tree(2, Tree(1), Tree(3))
```

Our claim that searching the tree can be performed in a logarithmic number of steps rests on the assumption that the tree is "balanced," i.e., that the left and the right subtree of every tree have approximately the same number of elements, so that each subtree contains about half the elements of its parent. But how can we be certain that the trees we construct will be balanced? Even if we start with a balanced tree, adding elements with `adjoin_set` may produce an unbalanced result. Since the position of a newly adjoined element depends on how the element compares with the items already in the set, we can expect that if we add elements "randomly" the tree will tend to be balanced on the average.

But this is not a guarantee. For example, if we start with an empty set and adjoin the numbers 1 through 7 in sequence we end up with a highly unbalanced tree in which all the left subtrees are empty, so it has no advantage over a simple ordered list. One way to solve this problem is to define an operation that transforms an arbitrary tree into a balanced tree with the same elements. We can perform this transformation after every few `adjoin_set` operations to keep our set in balance.

Intersection and union operations can be performed on tree-structured sets in linear time by converting them to ordered lists and back. The details are left as an exercise.

Python set implementation. The `set` type that is built into Python does not use any of these representations internally. Instead, Python uses a representation that gives constant-time membership tests and adjoin operations based on a technique called *hashing*, which is a topic for another course. Built-in Python sets cannot contain mutable data types, such as lists, dictionaries, or other sets. To allow for nested sets, Python also includes a built-in immutable `frozenset` class that shares methods with the `set` class but excludes mutation methods and operators.

3.4 Exceptions

Programmers must be always mindful of possible errors that may arise in their programs. Examples abound: a function may not receive arguments that it is designed to accept, a necessary resource may be missing, or a connection across a network may be lost. When designing a program, one must anticipate the exceptional circumstances that may arise and take appropriate measures to handle them.

There is no single correct approach to handling errors in a program. Programs designed to provide some persistent service like a web server should be robust to errors, logging them for later consideration but continuing to service new requests as long as possible. On the other hand, the Python interpreter handles errors by terminating immediately and printing an error message, so that programmers can address issues as soon as they arise. In any case, programmers must make conscious choices about how their programs should react to exceptional conditions.

Exceptions, the topic of this section, provides a general mechanism for adding error-handling logic to programs. *Raising an exception* is a technique for interrupting the normal flow of execution in a program, signaling that some exceptional circumstance has arisen, and returning directly to an enclosing part of the

program that was designated to react to that circumstance. The Python interpreter raises an exception each time it detects an error in an expression or statement. Users can also raise exceptions with `raise` and `assert` statements.

Raising exceptions. An exception is a object instance with a class that inherits, either directly or indirectly, from the `BaseException` class. The `assert` statement introduced in Chapter 1 raises an exception with the class `AssertionError`. In general, any exception instance can be raised with the `raise` statement. The general form of raise statements are described in the Python docs. The most common use of `raise` constructs an exception instance and raises it.

```
>>> raise Exception('An error occurred')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
Exception: an error occurred
```

When an exception is raised, no further statements in the current block of code are executed. Unless the exception is *handled* (described below), the interpreter will return directly to the interactive read-eval-print loop, or terminate entirely if Python was started with a file argument. In addition, the interpreter will print a *stack backtrace*, which is a structured block of text that describes the nested set of active function calls in the branch of execution in which the exception was raised. In the example above, the file name `<stdin>` indicates that the exception was raised by the user in an interactive session, rather than from code in a file.

Handling exceptions. An exception can be handled by an enclosing `try` statement. A `try` statement consists of multiple clauses; the first begins with `try` and the rest begin with `except`:

```
try:
    <try suite>
except <exception class> as <name>:
    <except suite>
...
```

The `<try suite>` is always executed immediately when the `try` statement is executed. Suites of the `except` clauses are only executed when an exception is raised during the course of executing the `<try suite>`. Each `except` clause specifies the particular class of exception to handle. For instance, if the `<exception class>` is `AssertionError`, then any instance of a class inheriting from `AssertionError` that is raised during the course of executing the `<try suite>` will be handled by the following `<except suite>`. Within the `<except suite>`, the identifier `<name>` is bound to the exception object that was raised, but this binding does not persist beyond the `<except suite>`.

For example, we can handle a `ZeroDivisionError` exception using a `try` statement that binds the name `x` to `0` when the exception is raised.

```
>>> try:
        x = 1/0
    except ZeroDivisionError as e:
```

```
        print('handling a', type(e))
        x = 0
handling a <class 'ZeroDivisionError'>
>>> x
0
```

A `try` statement will handle exceptions that occur within the body of a function that is applied (either directly or indirectly) within the `<try suite>`. When an exception is raised, control jumps directly to the body of the `<except suite>` of the most recent `try` statement that handles that type of exception.

```
>>> def invert(x):
        result = 1/x  # Raises a ZeroDivisionError if x is 0
        print('Never printed if x is 0')
        return result

>>> def invert_safe(x):
        try:
            return invert(x)
        except ZeroDivisionError as e:
            return str(e)

>>> invert_safe(2)
Never printed if x is 0
0.5
>>> invert_safe(0)
'division by zero'
```

This example illustrates that the `print` expression in `invert` is never evaluated, and instead control is transferred to the suite of the `except` clause in `handler`. Coercing the `ZeroDivisionError` `e` to a string gives the human-interpretable string returned by `handler`: `'division by zero'`.

3.4.1 Exception Objects

Exception objects themselves carry attributes, such as the error message stated in an `assert` statement and information about where in the course of execution the exception was raised. User-defined exception classes can carry additional attributes.

In Chapter 1, we implemented Newton's method to find the zeroes of arbitrary functions. The following example defines an exception class that returns the best guess discovered in the course of iterative improvement whenever a `ValueError` occurs. A math domain error (a type of `ValueError`) is raised when `sqrt` is applied to a negative number. This exception is handled by raising an `IterImproveError` that stores the most recent guess from Newton's method as an attribute.

First, we define a new class that inherits from `Exception`.

```
>>> class IterImproveError(Exception):
        def __init__(self, last_guess):
            self.last_guess = last_guess
```

Next, we define a version of `IterImprove`, our generic iterative improvement algorithm. This version handles any `ValueError` by raising an `IterImproveError` that stores the most recent guess. As before, `iter_improve` takes as arguments two functions, each of which takes a single numerical argument. The `update` function returns new guesses, while the `done` function returns a boolean indicating that improvement has converged to a correct value.

```
>>> def iter_improve(update, done, guess=1, max_updates=1000):
        k = 0
        try:
            while not done(guess) and k < max_updates:
                guess = update(guess)
                k = k + 1
            return guess
        except ValueError:
            raise IterImproveError(guess)
```

Finally, we define `find_root`, which returns the result of `iter_improve` applied to a Newton update function returned by `newton_update`, which is defined in Chapter 1 and requires no changes for this example. This version of `find_root` handles an `IterImproveError` by returning its last guess.

```
>>> def find_root(f, guess=1):
        def done(x):
            return f(x) == 0
        try:
            return iter_improve(newton_update(f), done, guess)
        except IterImproveError as e:
            return e.last_guess
```

Consider applying `find_root` to find the zero of the function $2x^2 + \sqrt{x}$. This function has a zero at `0`, but evaluating it on any negative number will raise a `ValueError`. Our Chapter 1 implementation of Newton's Method would raise that error and fail to return any guess of the zero. Our revised implementation returns the last guess found before the error.

```
>>> from math import sqrt
>>> find_root(lambda x: 2*x*x + sqrt(x))
-0.030211203830201594
```

While this approximation is still far from the correct answer of `0`, some applications would prefer this coarse approximation to a `ValueError`.

Exceptions are another technique that help us as programs to separate the concerns of our program into modular parts. In this example, Python's exception mechanism allowed us to separate the logic for iterative improvement, which appears unchanged in the suite of the `try` clause, from the logic for handling errors, which appears in `except` clauses. We will also find that exceptions are a very useful feature when implementing interpreters in Python.

3.5 Functional Programming

The software running on any modern computer is written in a variety of programming languages. There are physical languages, such as the machine languages for particular computers. These languages are concerned with the representation of data and control in terms of individual bits of storage and primitive machine instructions. The machine-language programmer is concerned with using the given hardware to erect systems and utilities for the efficient implementation of resource-limited computations. High-level languages, erected on a machine-language substrate, hide concerns about the representation of data as collections of bits and the representation of programs as sequences of primitive instructions. These languages have means of combination and abstraction, such as function definition, that are appropriate to the larger-scale organization of software systems.

In this section, we introduce a high-level programming language that encourages a functional style. Our object of study, Scheme, employs a very similar model of computation to Python's, but uses only expressions (no statements), specializes in symbolic computation, and primarily employs immutable values.

Scheme is a dialect of Lisp, the second-oldest programming language that is still widely used today (after Fortran). The community of Lisp programmers has continued to thrive for decades, and new dialects of Lisp such as Clojure have some of the fastest growing communities of developers of any modern programming language. To follow along with the examples in this text, you can download a Scheme interpreter.

3.5.1 Expressions

Scheme programs consist of expressions, which are either call expressions or special forms. A call expression consists of an operator expression followed by zero or more operand sub-expressions, as in Python. Both the operator and operand are contained within parentheses:

```
> (quotient 10 2)
5
```

Scheme exclusively uses prefix notation. Operators are often symbols, such as + and *. Call expressions can be nested, and they may span more than one line:

```
> (+ (* 3 5) (- 10 6))
19
> (+ (* 3
        (+ (* 2 4)
           (+ 3 5)))
     (+ (- 10 7)
        6))
57
```

As in Python, Scheme expressions may be primitives or combinations. Number literals are primitives, while call expressions are combined forms that include arbitrary sub-expressions. The evaluation procedure of call expressions matches that of Python: first the operator and operand expressions are evaluated, and then the function that is the value of the operator is applied to the arguments that are the values of the operands.

The `if` expression in Scheme is a *special form*, meaning that while it looks syntactically like a call expression, it has a different evaluation procedure. The general form of an `if` expression is:

```
(if <predicate> <consequent> <alternative>)
```

To evaluate an `if` expression, the interpreter starts by evaluating the `<predicate>` part of the expression. If the `<predicate>` evaluates to a true value, the interpreter then evaluates the `<consequent>` and returns its value. Otherwise it evaluates the `<alternative>` and returns its value.

Numerical values can be compared using familiar comparison operators, but prefix notation is used in this case as well:

```
> (>= 2 1)
#t
```

The boolean values `#t` (or `true`) and `#f` (or `false`) in Scheme can be combined with boolean special forms, which have evaluation procedures similar to those in Python.

- `(and <e1> ... <en>)` The interpreter evaluates the expressions `<e>` one at a time, in left-to-right order. If any `<e>` evaluates to `false`, the value of the `and` expression is `false`, and the rest of the `<e>`'s are not evaluated. If all `<e>`'s evaluate to true values, the value of the `and` expression is the value of the last one.
- `(or <e1> ... <en>)` The interpreter evaluates the expressions `<e>` one at a time, in left-to-right order. If any `<e>` evaluates to a true value, that value is returned as the value of the `or` expression, and the rest of the `<e>`'s are not evaluated. If all `<e>`'s evaluate to `false`, the value of the `or` expression is `false`.
- `(not <e>)` The value of a not expression is `true` when the expression `<e>` evaluates to `false`, and `false` otherwise.

3.5.2 Definitions

Values can be named using the `define` special form:

```
> (define pi 3.14)
> (* pi 2)
6.28
```

New functions (called *procedures* in Scheme) can be defined using a second version of the `define` special form. For example, to define squaring, we write:

```
(define (square x) (* x x))
```

The general form of a procedure definition is:

```
(define (<name> <formal parameters>) <body>)
```

The `<name>` is a symbol to be associated with the procedure definition in the environment. The `<formal parameters>` are the names used within the body of the procedure to refer to the corresponding arguments of the procedure. The `<body>` is an expression that will yield the value of the procedure application when the formal parameters are replaced by the actual arguments to which the procedure is applied. The `<name>` and the `<formal parameters>` are grouped within parentheses, just as they would be in an actual call to the procedure being defined.

Having defined square, we can now use it in call expressions:

```
> (square 21)
441

> (square (+ 2 5))
49

> (square (square 3))
81
```

User-defined functions can take multiple arguments and include special forms:

```
> (define (average x y)
    (/ (+ x y) 2))
> (average 1 3)
2
> (define (abs x)
    (if (< x 0)
        (- x)
```

```
        x))
> (abs -3)
3
```

Scheme supports local definitions with the same lexical scoping rules as Python. Below, we define an iterative procedure for computing square roots using nested definitions are recursion:

```
> (define (sqrt x)
    (define (good-enough? guess)
      (< (abs (- (square guess) x)) 0.001))
    (define (improve guess)
      (average guess (/ x guess)))
    (define (sqrt-iter guess)
      (if (good-enough? guess)
          guess
          (sqrt-iter (improve guess))))
    (sqrt-iter 1.0))
> (sqrt 9)
3.00009155413138
```

Anonymous functions are created using the `lambda` special form. `Lambda` is used to create procedures in the same way as `define`, except that no name is specified for the procedure:

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

The resulting procedure is just as much a procedure as one that is created using `define`. The only difference is that it has not been associated with any name in the environment. In fact, the following expressions are equivalent:

```
> (define (plus4 x) (+ x 4))
> (define plus4 (lambda (x) (+ x 4)))
```

Like any expression that has a procedure as its value, a lambda expression can be used as the operator in a call expression:

```
> ((lambda (x y z) (+ x y (square z))) 1 2 3)
12
```

3.5.3 Compound values

Pairs are built into the Scheme language. For historical reasons, pairs are created with the `cons` built-in function, and the elements of a pair are accessed with `car` and `cdr`:

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

Recursive lists are also built into the language, using pairs. A special value denoted `nil` or `'()` represents the empty list. Recursive lists are written as values contained within parentheses:

```
> (cons 1
        (cons 2
              (cons 3
                    (cons 4 nil))))
(1 2 3 4)
> (list 1 2 3 4)
(1 2 3 4)
> (define one-through-four (list 1 2 3 4))
> (car one-through-four)
1
> (cdr one-through-four)
(2 3 4)
> (car (cdr one-through-four))
2
> (cons 10 one-through-four)
(10 1 2 3 4)
> (cons 5 one-through-four)
(5 1 2 3 4)
```

Whether a list is empty can be determined using the primitive `null?` predicate. Using it, we can define the standard sequence operations for computing `length` and selecting elements:

```
> (define (length items)
    (if (null? items)
        0
        (+ 1 (length (cdr items)))))
> (define (getitem items n)
    (if (= n 0)
        (car items)
        (getitem (cdr items) (- n 1))))
> (define squares (list 1 4 9 16 25))
> (length squares)
5
> (getitem squares 3)
16
```

3.5.4 Symbolic Data

All the compound data objects we have used so far were constructed ultimately from numbers. One of Scheme's strengths is working with arbitrary symbols as data.

In order to manipulate symbols we need a new element in our language: the ability to *quote* a data object. Suppose we want to construct the list `(a b)`. We can't accomplish this with `(list a b)`, because this expression constructs a list of the values of `a` and `b` rather than the symbols themselves. In Scheme, we refer to the symbols `a` and `b` rather than their values by preceding them with a single quotation mark:

```
> (define a 1)
> (define b 2)
> (list a b)
(1 2)
> (list 'a 'b)
(a b)
> (list 'a b)
(a 2)
```

In dialects of Lisp (and Scheme is such a dialect), any expression that is not evaluated is said to be *quoted*. This notion of quotation is derived from a classic philosophical distinction between a thing, such as a dog, which runs around and barks, and the word "dog" that is a linguistic construct for designating such things. When we use "dog" in quotation marks, we do not refer to some dog in particular but instead to a word. In language, quotation allow us to talk about language itself, and so it is in Scheme:

```
> (list 'define 'list)
(define list)
```

Quotation also allows us to type in compound objects, using the conventional printed representation for lists:

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

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

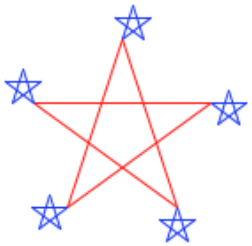
The full Scheme language contains additional features, such as mutation operations, vectors, and maps. However, the subset we have introduced so far provides a rich functional programming language capable of implementing many of the ideas we have discussed so far in this text.

3.5.5 Turtle graphics

The implementation of Scheme that serves as a companion to this text includes Turtle graphics, an illustrating environment developed as part of the Logo language (another Lisp dialect). This turtle begins in the center of a canvas, moves and turns based on procedures, and draws lines behind it as it moves. While the turtle was invented to engage children in the act of programming, it remains an entertaining graphical tool for even advanced programmers.

At any moment during the course of executing a Scheme program, the turtle has a position and heading on the canvas. Single-argument procedures such as `forward` and `right` change the position and heading of the turtle. Common procedures have abbreviations: `forward` can also be called as `fd`, etc. The `begin` special form in Scheme allows a single expression to include multiple sub-expressions. This form is useful for issuing multiple commands:

```
> (define (repeat k fn) (if (> k 0)
                            (begin (fn) (repeat (- k 1) fn))
                            nil))
> (repeat 5
          (lambda () (fd 100)
                     (repeat 5
                             (lambda () (fd 20) (rt 144)))
                     (rt 144)))
nil
```



The full repertoire of Turtle procedures is also built into Python as the turtle library module.

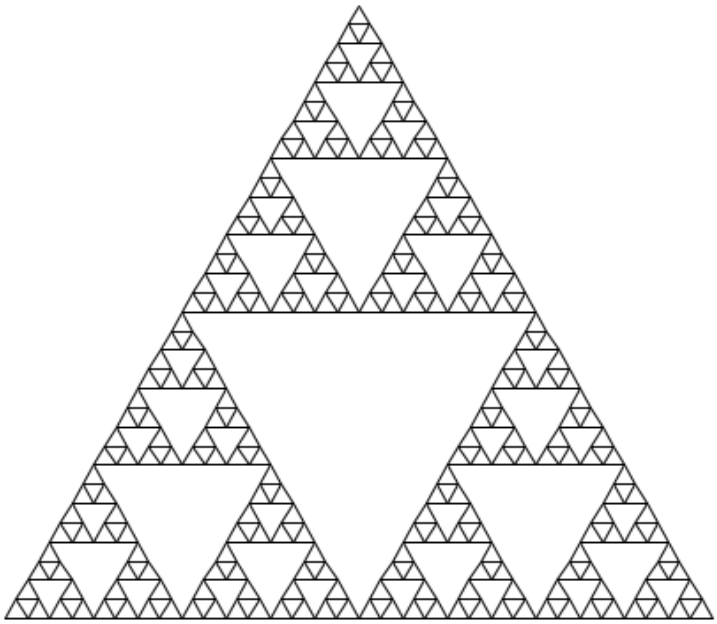
As a final example, Scheme can express recursive drawings using its turtle graphics in a remarkably compact form. Sierpinski's triangle is a fractal that draws each triangle as three neighboring triangles that have vertexes at the midpoints of the legs of the triangle that contains them. It can be drawn to a finite recursive depth by this Scheme program:

```
> (define (repeat k fn)
    (if (> k 0)
        (begin (fn) (repeat (- k 1) fn))
        nil))

> (define (tri fn)
    (repeat 3 (lambda () (fn) (lt 120))))

> (define (sier d k)
    (tri (lambda ()
           (if (= k 1) (fd d) (leg d k)))))
```

```
> (define (leg d k)
    (sier (/ d 2) (- k 1))
    (penup)
    (fd d)
    (pendown))
```

The `triangle` procedure is a general method for repeating a drawing procedure three times with a left turn following each repetition. The `sier` procedure takes a length `d` and a recursive depth `k`. It draws a plain triangle if the depth is `1`, and otherwise draws a triangle made up of calls to `leg`. The `leg` procedure draws a single leg of a recursive Sierpinski triangle by a recursive call to `sier` that fills the first half of the length of the leg, then by moving the turtle to the next vertex. The procedures `penup` and `pendown` stop the turtle from drawing as it moves by lifting its pen up and the placing it down again. The mutual recursion between `sier` and `leg` yields this result:

```
> (sier 400 6)
```



3.6 Interpreters for Languages with Combination

We now embark on a tour of the technology by which languages are established in terms of other languages. *Metalinguistic abstraction* -- establishing new languages -- plays an important role in all branches of engineering design. It is particularly important to computer programming, because in programming not only can we formulate new languages but we can also implement these languages by constructing interpreters. An interpreter for a programming language is a function that, when applied to an expression of the language, performs the actions required to evaluate that expression.

We will first define an interpreter for a language that is a limited subset of Scheme, called Calculator. Then, we will develop a sketch of an interpreter for Scheme as a whole. The interpreter we create will be complete in the sense that it will allow us to write fully general programs in Scheme. To do so, it will implement the environment model of evaluation that we developed in Chapter 1.

Many of the examples in this section are contained in the companion Scheme-Syntax Calculator example, as they are too complex to fit naturally in the format of this text.

3.6.1 A Scheme-Syntax Calculator

The Scheme-Syntax Calculator (or simply Calculator) is an expression language for the arithmetic operations of addition, subtraction, multiplication, and division. Calculator shares Scheme's call expression syntax and operator behavior. Addition (`+`) and multiplication (`*`) operations each take an arbitrary number of arguments:

```
> (+ 1 2 3 4)
10
> (+)
0
> (* 1 2 3 4)
24
> (*)
1
```

Subtraction (`-`) has two behaviors. With one argument, it negates the argument. With at least two arguments, it subtracts all but the first from the first. Division (`/`) takes exactly two arguments:

```
> (- 10 1 2 3)
4
> (- 3)
-3
> (/ 15 12)
1.25
> (/ 15 5)
3
```

A call expression is evaluated by evaluating its operand sub-expressions, then applying the operator to the resulting arguments:

```
> (- 100 (* 7 (+ 8 (/ -12 -3))))
16.0
```

We will implement an interpreter for the Calculator language in Python. That is, we will write a Python program that takes string lines as input and returns the result of evaluating those lines as a Calculator expression. Our interpreter will raise an appropriate exception if the calculator expression is not well formed.

3.6.2 Expression Trees

Until this point in the course, expression trees have been conceptual entities to which we have referred in describing the process of evaluation; we have never before explicitly represented expression trees as data in our programs. In order to write an interpreter, we must operate on expressions as data.

A primitive expression is just a number or a string in Calculator: either an `int` or `float` or an operator symbol. All combined expressions are call expressions. A call expression is a Scheme list with a first element (the operator) followed by zero or more operand expressions.

Scheme Pairs. In Scheme, lists are nested pairs, but not all pairs are lists. To represent Scheme pairs and lists in Python, we will define a class `Pair` that is similar to the `Rlist` class earlier in the chapter. The implementation appears in scheme_reader.

The empty list is represented by an object called `nil`, which is an instance of the class `nil`. We assume that only one `nil` instance will ever be created.

The `Pair` class and `nil` object are Scheme values represented in Python. They have `repr` strings that are Python expressions and `str` strings that are Scheme expressions.

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

They implement the basic Python sequence interface of length and element selection, as well as a `map` method that returns a Scheme list.

```
>>> len(s)
2
>>> s[1]
2
>>> print(s.map(lambda x: x+4))
(5 6)
```

Trees. Trees are represented in Scheme by allowing the elements of a Scheme list to be Scheme lists. The nested Scheme expression `(+ (* 3 4) 5)` is a tree, and it is represented as

```
>>> exp = Pair('+', Pair(Pair('*', Pair(3, Pair(4, nil))), Pair(5, nil)))
>>> print(exp)
(+ (* 3 4) 5)
>>> print(exp.second.first)
(* 3 4)
>>> exp.second.first.second.first
3
```

This example demonstrates that all Calculator expressions are nested Scheme lists. Our Calculator interpreter will read in nested Scheme lists, convert them into expression trees represented as nested `Pair` instances (*Parsing expressions* below), and then evaluate the expression trees to produce values (*Calculator evaluation* below).

3.6.3 Parsing Expressions

Parsing is the process of generating expression trees from raw text input. A parser is a composition of two components: a lexical analyzer and a syntactic analyzer. First, the *lexical analyzer* partitions the input string into *tokens*, which are the minimal syntactic units of the language such as names and symbols. Second, the *syntactic analyzer* constructs an expression tree from this sequence of tokens. The sequence of tokens produced by the lexical analyzer is consumed by the syntactic analyzer.

Lexical analysis. The component that interprets a string as a token sequence is called a *tokenizer* or *lexical analyzer*. In our implementation, the tokenizer is a function called `tokenize_line` in scheme_tokens. Scheme tokens are delimited by white space, parentheses, dots, or single quotation marks. Delimiters are tokens, as are symbols and numerals. The tokenizer analyzes a line character by character, validating the format of symbols and numerals.

Tokenizing a well-formed Calculator expression separates all symbols and delimiters, but identifies multi-character numbers (e.g., `2.3`) and converts them into numeric types.

```
>>> tokenize_line('(+ 1 (* 2.3 45))')
['(', '+', 1, '(', '*', 2.3, 45, ')', ')']
```

Lexical analysis is an iterative process, and it can be applied to each line of an input program in isolation.

Syntactic analysis. The component that interprets a token sequence as an expression tree is called a *syntactic analyzer*. Syntactic analysis is a tree-recursive process, and it must consider an entire expression that may span multiple lines.

Syntactic analysis is implemented by the `scheme_read` function in scheme_reader. It is tree-recursive because analyzing a sequence of tokens often involves analyzing a subsequence of those tokens into a subexpression, which itself serves as a branch (e.g., operand) of a larger expression tree. Recursion generates the hierarchical structures consumed by the evaluator.

The `scheme_read` function expects its input `src` to be a `Buffer` instance that gives access to a sequence of tokens. A `Buffer`, defined in the buffer module, collects tokens that span multiple lines into a single object that can be analyzed syntactically.

```
>>> lines = ['(+ 1', '   (* 2.3 45))']
>>> expression = scheme_read(Buffer(tokenize_lines(lines)))
>>> expression
Pair('+', Pair(1, Pair(Pair('*', Pair(2.3, Pair(45, nil))), nil)))
```

```
>>> print(expression)
(+ 1 (* 2.3 45))
```

The `scheme_read` function first checks for various base cases, including empty input (which raises an end-of-file exception, called `EOFError` in Python) and primitive expressions. A recursive call to `read_tail` is invoked whenever a `(` token indicates the beginning of a list.

The `read_tail` function continues to read from the same input `src`, but expects to be called after a list has begun. Its base cases are an empty input (an error) or a closing parenthesis that terminates the list. Its recursive call reads the first element of the list with `scheme_read`, reads the rest of the list with `read_tail`, and then returns a list represented as a `Pair`.

This implementation of `scheme_read` can read well-formed Scheme lists, which are all we need for the Calculator language. Parsing dotted lists and quoted forms is left as an exercise.

Informative syntax errors improve the usability of an interpreter substantially. The `SyntaxError` exceptions that are raised include a description of the problem encountered.

3.6.4 Calculator Evaluation

The scalc module implements an evaluator for the Calculator language. The `calc_eval` function takes an expression as an argument and returns its value. Definitions of the helper functions `simplify`, `reduce`, and `scheme_list` appear in the model and are used below.

For Calculator, the only two legal syntactic forms of expressions are numbers and call expressions, which are `Pair` instances representing well-formed Scheme lists. Numbers are *self-evaluating*; they can be returned directly from `calc_eval`. Call expressions require function application.

```
>>> def calc_eval(exp):
        """Evaluate a Calculator expression."""
        if type(exp) in (int, float):
            return simplify(exp)
        elif isinstance(exp, Pair):
            arguments = exp.second.map(calc_eval)
            return simplify(calc_apply(exp.first, arguments))
        else:
            raise TypeError(exp + ' is not a number or call expression')
```

Call expressions are evaluated by first recursively mapping the `calc_eval` function to the list of operands, which computes a list of `arguments`. Then, the operator is applied to those arguments in a second function, `calc_apply`.

The Calculator language is simple enough that we can easily express the logic of applying each operator in the body of a single function. In `calc_apply`, each conditional clause corresponds to applying one operator.

```
>>> def calc_apply(operator, args):
        """Apply the named operator to a list of args."""
        if operator == '+':
            return reduce(add, args, 0)
        elif operator == '-':
            if len(args) == 0:
                raise TypeError(operator + 'requires at least 1 argument')
            elif len(args) == 1:
                return -args[0]
            else:
                return reduce(sub, args.second, args.first)
        elif operator == '*':
            return reduce(mul, args, 1)
        elif operator == '/':
            if len(args) != 2:
                raise TypeError(operator + ' requires exactly 2 arguments')
            numer, denom = args
            return numer/denom
```

Above, each suite computes the result of a different operator or raises an appropriate `TypeError` when the wrong number of arguments is given. The `calc_apply` function can be applied directly, but it must be passed a list of *values* as arguments rather than a list of operand expressions.

```
>>> calc_apply('+', scheme_list(1, 2, 3))
6
>>> calc_apply('-', scheme_list(10, 1, 2, 3))
4
>>> calc_apply('*', nil)
1
>>> calc_apply('*', scheme_list(1, 2, 3, 4, 5))
120
>>> calc_apply('/', scheme_list(40, 5))
8.0
```

The role of `calc_eval` is to make proper calls to `calc_apply` by first computing the value of operand sub-expressions before passing them as arguments to `calc_apply`. Thus, `calc_eval` can accept a nested expression.

```
>>> print(exp)
(+ (* 3 4) 5)
>>> calc_eval(exp)
17
```

The structure of `calc_eval` is an example of dispatching on type: the form of the expression. The first form of expression is a number, which requires no additional evaluation step. In general, primitive expressions that do not require an additional evaluation step are called *self-evaluating*. The only self-evaluating

expressions in our Calculator language are numbers, but a general programming language might also include strings, boolean values, etc.

Read-eval-print loops. A typical approach to interacting with an interpreter is through a read-eval-print loop, or REPL, which is a mode of interaction that reads an expression, evaluates it, and prints the result for the user. The Python interactive session is an example of such a loop.

An implementation of a REPL can be largely independent of the interpreter it uses. The function `read_eval_print_loop` below buffers input from the user, constructs an expression using the language-specific `scheme_read` function, then prints the result of applying `calc_eval` to that expression.

```
>>> def read_eval_print_loop():
        """Run a read-eval-print loop for calculator."""
        while True:
            src = buffer_input()
            while src.more_on_line:
                expression = scheme_read(src)
                print(calc_eval(expression))
```

This version of `read_eval_print_loop` contains all of the essential components of an interactive interface. An example session would look like:

```
> (* 1 2 3)
6
> (+)
0
> (+ 2 (/ 4 8))
2.5
> (+ 2 2) (* 3 3)
4
9
> (+ 1
     (- 23)
     (* 4 2.5))
-12
```

This loop implementation has no mechanism for termination or error handling. We can improve the interface by reporting errors to the user. We can also allow the user to exit the loop by signalling a keyboard interrupt (`Control-C` on UNIX) or end-of-file exception (`Control-D` on UNIX). To enable these improvements, we place the original suite of the `while` statement within a `try` statement. The first `except` clause handles `SyntaxError` and `ValueError` exceptions raised by `scheme_read` as well as `TypeError` and `ZeroDivisionError` exceptions raised by `calc_eval`.

```
>>> def read_eval_print_loop():
        """Run a read-eval-print loop for calculator."""
        while True:
            try:
```

```
            src = buffer_input()
            while src.more_on_line:
                expression = scheme_read(src)
                print(calc_eval(expression))
        except (SyntaxError, TypeError, ValueError, ZeroDivisionError) as err:
            print(type(err).__name__ + ':', err)
        except (KeyboardInterrupt, EOFError):  # <Control>-D, etc.
            print('Calculation completed.')
            return
```

This loop implementation reports errors without exiting the loop. Rather than exiting the program on an error, restarting the loop after an error message lets users revise their expressions. Upon importing the `readline` module, users can even recall their previous inputs using the up arrow or `Control-P`. The final result provides an informative error reporting interface:

```
> )
SyntaxError: unexpected token: )
> 2.3.4
ValueError: invalid numeral: 2.3.4
> +
TypeError: + is not a number or call expression
> (/ 5)
TypeError: / requires exactly 2 arguments
> (/ 1 0)
ZeroDivisionError: division by zero
```

As we generalize our interpreter to new languages other than Calculator, we will see that the `read_eval_print_loop` is parameterized by a parsing function, an evaluation function, and the exception types handled by the `try` statement. Beyond these changes, all REPLs can be implemented using the same structure.

3.7 Interpreters for Languages with Abstraction

The Calculator language provides a means of combination through nested call expressions. However, there is no way to define new operators, give names to values, or express general methods of computation. Calculator does not support abstraction in any way. As a result, it is not a particularly powerful or general programming language. We now turn to the task of defining a general programming language that supports abstraction by binding names to values and defining new operations.

Unlike the previous section, which presented a complete interpreter as Python source code, this section takes a descriptive approach. The companion project asks you to implement the ideas presented here by building a fully functional Scheme interpreter.

3.7.1 Structure

This section describes the general structure of a Scheme interpreter. Completing that project will produce a working implementation of the interpreter described here.

An interpreter for Scheme can share much of the same structure as the Calculator interpreter. A parser produces an expression that is interpreted by an evaluator. The evaluation function inspects the form of an expression, and for call expressions it calls a function to apply a procedure to some arguments. Much of the difference in evaluators is associated with special forms, user-defined functions, and implementing the environment model of computation.

Parsing. The scheme_reader and scheme_tokens modules from the Calculator interpreter are nearly sufficient to parse any valid Scheme expression. However, it does not yet support quotation or dotted lists. A full Scheme interpreter should be able to parse the following input expression.

```
>>> read_line("(car '(1 . 2))")
Pair('car', Pair(Pair('quote', Pair(Pair(1, 2), nil)), nil))
```

Your first task in implementing the Scheme interpreter will be to extend scheme_reader to correctly parse dotted lists and quotation.

Evaluation. Scheme is evaluated one expression at a time. A skeleton implementation of the evaluator is defined in `scheme.py` of the companion project. Each expression returned from `scheme_read` is passed to the `scheme_eval` function, which evaluates an expression `expr` in the current environment `env`.

The `scheme_eval` function evaluates the different forms of expressions in Scheme: primitives, special forms, and call expressions. The form of a combination in Scheme can be determined by inspecting its first element. Each special form has its own evaluation rule. A simplified implementation of `scheme_eval` appears below. Some error checking and special form handling has been removed in order to focus our discussion. A complete implementation appears in the companion project.

```
>>> def scheme_eval(expr, env):
        """Evaluate Scheme expression expr in environment env."""
        if scheme_symbolp(expr):
            return env[expr]
        elif scheme_atomp(expr):
            return expr
        first, rest = expr.first, expr.second
        if first == "lambda":
            return do_lambda_form(rest, env)
        elif first == "define":
            do_define_form(rest, env)
            return None
        else:
            procedure = scheme_eval(first, env)
            args = rest.map(lambda operand: scheme_eval(operand, env))
            return scheme_apply(procedure, args, env)
```

Procedure application. The final case above invokes a second process, procedure application, that is implemented by the function `scheme_apply`. The procedure application process in Scheme is considerably more general than the `calc_apply` function in Calculator. It applies two kinds of arguments: a `PrimtiveProcedure` or a `LambdaProcedure`. A `PrimitiveProcedure` is implemented in Python; it has an instance attribute `fn` that is bound to a Python function. In addition, it may or may not require access to the current environment. This Python function is called whenever the procedure is applied.

A `LambdaProcedure` is implemented in Scheme. It has a `body` attribute that is a Scheme expression, evaluated whenever the procedure is applied. To apply the procedure to a list of arguments, the body expression is evaluated in a new environment. To construct this environment, a new frame is added to the environment, in which the formal parameters of the procedure are bound to the arguments. The body is evaluated using `scheme_eval`.

Eval/apply recursion. The functions that implement the evaluation process, `scheme_eval` and `scheme_apply`, are mutually recursive. Evaluation requires application whenever a call expression is encountered. Application uses evaluation to evaluate operand expressions into arguments, as well as to evaluate the body of user-defined procedures. The general structure of this mutually recursive process appears in interpreters quite generally: evaluation is defined in terms of application and application is defined in terms of evaluation.

This recursive cycle ends with language primitives. Evaluation has a base case that is evaluating a primitive expression. Some special forms also constitute base cases without recursive calls. Function application has a base case that is applying a primitive procedure. This mutually recursive structure, between an eval function that processes expression forms and an apply function that processes functions and their arguments, constitutes the essence of the evaluation process.

3.7.2 Environments

Now that we have described the structure of our Scheme interpreter, we turn to implementing the `Frame` class that forms environments. Each `Frame` instance represents an environment in which symbols are bound to values. A frame has a dictionary of `bindings`, as well as a `parent` frame that is `None` for the global frame.

Bindings are not accessed directly, but instead through two `Frame` methods: `lookup` and `define`. The first implements the look-up procedure of the environment model of computation described in Chapter 1. A symbol is matched against the `bindings` of the current frame. If it is found, the value to which it is bound is returned. If it is not found, look-up proceeds to the `parent` frame. On the other hand, the `define` method always binds a symbol to a value in the current frame.

The implementation of `lookup` and the use of `define` are left as exercises. As an illustration of their use, consider the following example Scheme program:

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

```
> (factorial 5)
120
```

The first input expression is a `define` special form, evaluated by the `do_define_form` Python function. Defining a function has several steps:

1. Check the format of the expression to ensure that it is a well-formed Scheme list with at least two elements following the keyword `define`.
2. Analyze the first element, in this case a `Pair`, to find the function name `factorial` and formal parameter list `(n)`.
3. Create a `LambdaProcedure` with the supplied formal parameters, body, and parent environment.
4. Bind the symbol `factorial` to this function, in the first frame of the current environment. In this case, the environment consists only of the global frame.

The second input is a call expression. The `procedure` passed to `scheme_apply` is the `LambdaProcedure` just created and bound to the symbol `factorial`. The `args` passed is a one-element Scheme list `(5)`. To apply the procedure, a new frame is created that extends the global frame (the parent environment of the `factorial` procedure). In this frame, the symbol `n` is bound to the value `5`. Then, the body of `factorial` is evaluated in that environment, and its value is returned.

3.7.3 Data as Programs

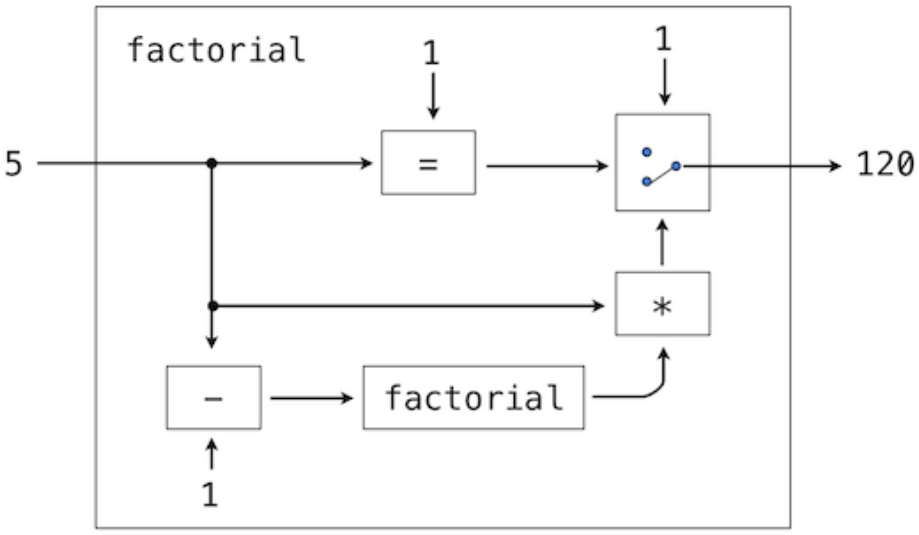
In thinking about a program that evaluates Scheme expressions, an analogy might be helpful. One operational view of the meaning of a program is that a program is a description of an abstract machine. For example, consider again this procedure to compute factorials:

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

We could express an equivalent program in Python as well, using a conditional expression.

```
>>> def factorial(n):
        return 1 if n == 1 else n * factorial(n - 1)
```

We may regard this program as the description of a machine containing parts that decrement, multiply, and test for equality, together with a two-position switch and another factorial machine. (The factorial machine is infinite because it contains another factorial machine within it.) The figure below is a flow diagram for the factorial machine, showing how the parts are wired together.



In a similar way, we can regard the Scheme interpreter as a very special machine that takes as input a description of a machine. Given this input, the interpreter configures itself to emulate the machine described. For example, if we feed our evaluator the definition of factorial the evaluator will be able to compute factorials.

From this perspective, our Scheme interpreter is seen to be a universal machine. It mimics other machines when these are described as Scheme programs. It acts as a bridge between the data objects that are manipulated by our programming language and the programming language itself. Image that a user types a Scheme expression into our running Scheme interpreter. From the perspective of the user, an input expression such as `(+ 2 2)` is an expression in the programming language, which the interpreter should evaluate. From the perspective of the Scheme interpreter, however, the expression is simply a sentence of words that is to be manipulated according to a well-defined set of rules.

That the user's programs are the interpreter's data need not be a source of confusion. In fact, it is sometimes convenient to ignore this distinction, and to give the user the ability to explicitly evaluate a data object as an expression. In Scheme, we use this facility whenever employing the `run` procedure. Similar functions exist in Python: the `eval` function will evaluate a Python expression and the `exec` function will execute a Python statement. Thus,

```
>>> eval('2+2')
4
```

and

```
>>> 2+2
4
```

both return the same result. Evaluating expressions that are constructed as a part of execution is a common and powerful feature in dynamic programming languages. In few languages is this practice as

common as in Scheme, but the ability to construct and evaluate expressions during the course of execution of a program can prove to be a valuable tool for any programmer.