Tree visitors in Clojure

Update the Java Visitor pattern with functional zippers

Skill Level: Intermediate

Alex Miller (alex@puredanger.com)
Senior Engineer
Revelytix

20 Sep 2011

JVM language explorer Alex Miller has recently discovered the benefits of implementing the Visitor pattern using Clojure, a functional Lisp variant for the Java Virtual Machine. In this article, he revisits the Visitor pattern, first demonstrating its use in traversing tree data in Java programs, then rewriting it with the addition of Clojure's functional zippers.

I've used trees of domain objects in my Java applications for many years. More recently, I've been using them in Clojure. In each case, I've found that the Visitor pattern is a reliable tool for manipulating trees of data. But there are differences in how the pattern works in a functional versus object-oriented language, and in the results it yields.

About Clojure
Clojure is a dynamic and functional programming language variant of Lisp, written specifically for the JVM. Learn more about Clojure on developerWorks:

- "The Clojure programming language"
- "Clojure and concurrency"
- "Solving the Expression Problem with Clojure 1.2"
- "Using CouchDB with Clojure"

In this article, I revisit domain trees and the Visitor pattern for the Java language, then walk through several visitor implementations using Clojure. Being a functional language, Clojure brings new tricks to data query and manipulation. In particular, I've found that integrating functional zippers into the Visitor pattern yields efficiency benefits, which I explore.

Manipulating symbolic trees

One example of a symbolic tree is a representation of an SQL query plan, which has operations like filter, join, union, and project. These operations work together to form a series of computations from source data to produce the result of a query.

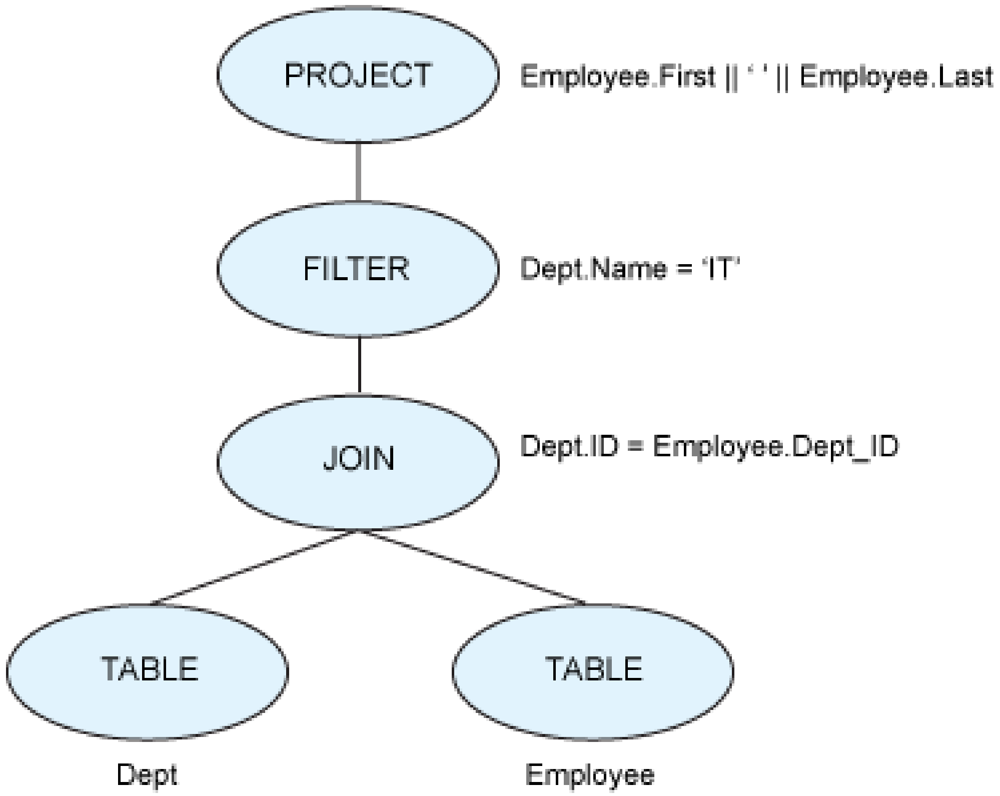
For example, the query in Listing 1 describes a join of the Dept and Employee tables. The query filters some results where the department name is "IT" and returns a concatenation of an employee’s first and last names. Its result should be the full names of all employees in the IT department.

Listing 1. Example SQL query

```
SELECT Employee.First || ' ' || Employee.Last
FROM Dept, Employee
WHERE Dept.ID = Employee.Dept_ID
  AND Dept.Name = 'IT'
```

We can then examine the equivalent symbolic tree representation of this query plan in Figure 1. Conceptually, rows of relational data (tuples) flow through the operation nodes in the query plan from bottom to top. The final results of the query are then pulled from the top.

Figure 1. Symbolic tree representation of the SQL query


PROJECT
Employee.First || ' ' || Employee.Last
FILTER
Dept.Name = 'IT'
JOIN
Dept.ID = Employee.Dept_ID
TABLE
TABLE
Dept
Employee

When using a tree of nodes like this, we need to perform operations on the tree as a whole. Possible operations include:

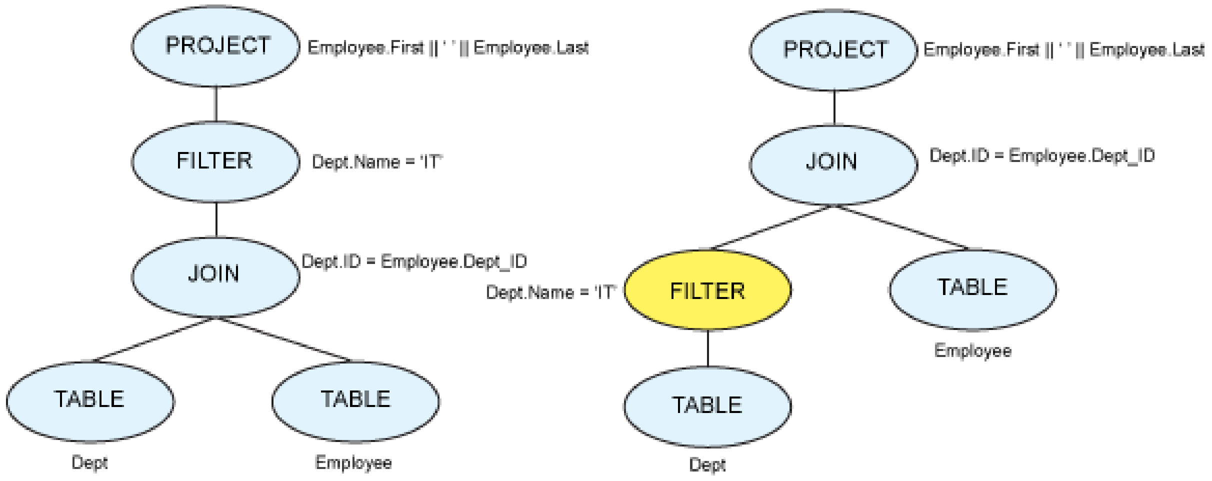
- Collect all tables referenced in the query plan.
- Collect all columns used in a subtree.
- Find the top join node in a query plan.
- Match a pattern in the tree and modify the tree.

The last operation is particularly useful because it lets us perform symbolic manipulations on the tree. Using the Filter Pushdown pattern, we would first match a pattern in the graph, then modify the tree at the point of the match. Here's the Filter Pushdown pattern to match:

- Filter node F directly above a Join node J.
- F has a criteria that only involves a single table.
- J is an inner join.

We could then apply a tree manipulation to move the Filter node underneath the Join node toward the related Table node. Manipulations like this (backed by the proper relational theory) are at the heart of database query optimizers. The resulting tree is shown in Figure 2:

Figure 2. Result of applying the Filter Pushdown optimization

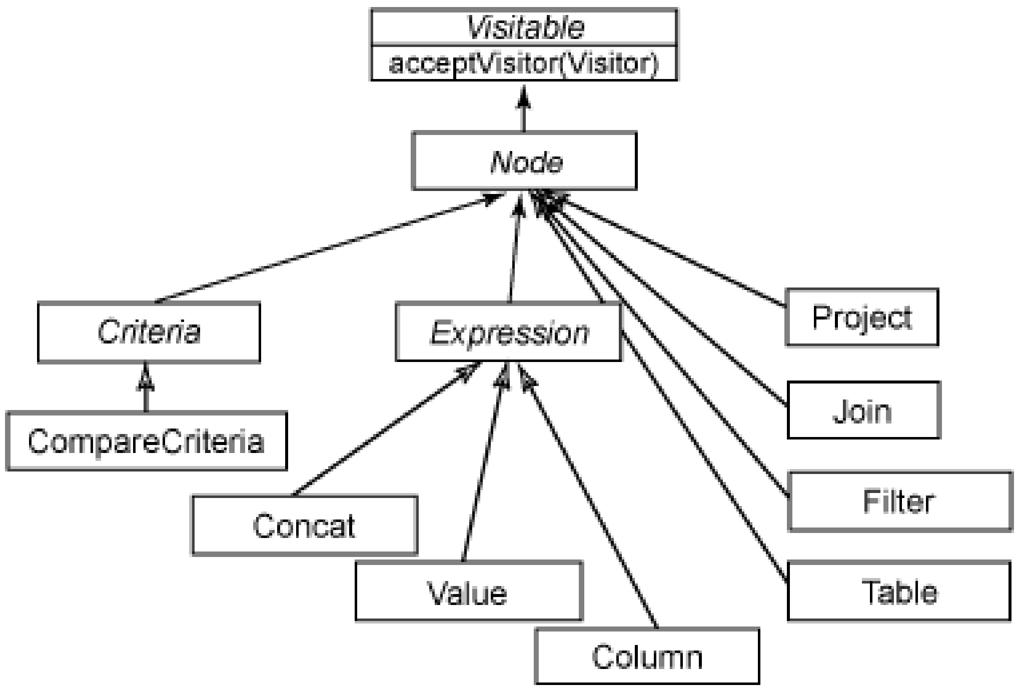

PROJECT
Employee.First || ' ' || Employee.Last
FILTER
Dept.Name = 'IT'
JOIN
Dept.ID = Employee.Dept_ID
TABLE
Dept
TABLE
Employee
PROJECT
Employee.First || ' ' || Employee.Last
JOIN
Dept.ID = Employee.Dept_ID
Dept.Name = 'IT'
FILTER
TABLE
Employee
TABLE
Dept

The Visitor pattern is often used to separate a data structure (the tree, in this case) from the algorithms that operate over the data structure. Later in this article, I'll demonstrate both an object-oriented implementation in Java code and a functional variant in Clojure.

Visitors in the Java language

If we want to implement the Visitor pattern in Java, we need first to represent our nodes as Java classes. We can build a basic hierarchy as shown in Figure 3. Most of these classes are simple data holders. For example, the `Join` class contains a left-join expression, a right-join expression, a join type, and a join criteria.

Figure 3. Java domain classes



Note that all of the objects in the domain hierarchy implement the `Visitable` interface and the `acceptVisitor` method, looking like so:

```
public void acceptVisitor(Visitor visitor) {
  visitor.visit(this);
}
```

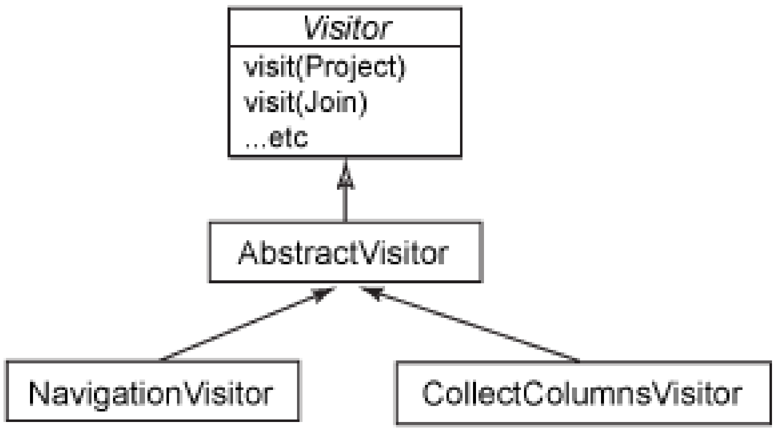
This `acceptVisitor` method implements *double dispatch*, allowing the choice of method call to depend not just on the concrete object type, but also on the visitor type.

The visitor classes are shown in Figure 4. The base `Visitor` interface contains a visit method for every concrete type in the domain. The `AbstractVisitor` implements empty versions of all these methods to make writing concrete visitors easier.

Visitor navigation

In addition to visiting each node, the visitor must decide which nodes should be visited as children of the current node. It is possible to include the navigation in each visitor, but it's better to separate the concerns of navigation and operation. In fact, you can think of finding children on a per-type basis as a visitor operation that can be encapsulated in a visitor itself. `NavigationVisitor` captures the tree navigation operation and lets a more lightweight visitor come along for the ride.

Figure 4. Visitor interfaces



Take a look at `NavigationVisitor`'s methods:

```
public void visit(Join j) {
    visitor.visit(j);
    j.getCriteria().acceptVisitor(this);
    j.getLeft().acceptVisitor(this);
    j.getRight().acceptVisitor(this);
  }
```

`NavigationVisitor` defers to another visitor instance but embeds the child navigation. An example visitor might collect all `Column` instances in the entire tree and would look like Listing 2:

Listing 2. CollectColumnsVisitor

```
package visitors;

import java.util.HashSet;
import java.util.Set;

import visitors.domain.Column;

public class CollectColumnsVisitor extends AbstractVisitor {
  private final Set<Column> columns = new HashSet<Column>();

  public Set<Column> getColumns() {
    return this.columns;
  }

  @Override
  public void visit(Column c) {
    columns.add(c);
  }
}
```

When this visitor finds a `Column` in the tree, it stores that into the set of all the columns seen so far. Once the visitor completes, we can retrieve the full set of `Columns`, as demonstrated in Listing 3:

Listing 3. Calling the CollectColumnsVisitor

```
Node tree = createTree();
CollectColumnsVisitor colVisitor = new CollectColumnsVisitor();
NavigationVisitor v = new NavigationVisitor(colVisitor);
tree.acceptVisitor(v);
System.out.println("columns = " + colVisitor.getColumns());
```

Once you have this basic structure in place, you can add any of the following optimizations:

- Navigation visitors that do pre, post, or custom navigation
- Avoid walking the entire tree
- Mutation of the tree while visiting

Incidental complexity in Java visitors

The Visitor pattern in the Java language partially addresses the classic *expression problem*. Coined by Philip Wadler, the expression problem defines a program’s data as a set of cases (types) and a set of operations over those cases. Imagine these as the two dimensions of a table. Can you then add new data types and new operations without recompiling and retain static types?

The Visitor pattern creates a scenario where adding operations (new visitors) over the set of existing data types is easy. Adding new data types (classes) with visitors is difficult, however, as the Visitor pattern requires a `visit()` method for all concrete types. You can alleviate this somewhat by having an abstract superclass that contains empty implementations of all methods for all visitors. In this case, you can modify just the abstract class and the code will compile. Visitors don’t meet the standard of no recompilation but they do minimize the changes required to add a new operation. If you also consider that adding a new type happens much less often than adding a new operation, the overall pattern makes good compromises: certainly better than encoding a new operation into a new method on all concrete types.

While implementing the Visitor pattern in the Java language allows us to satisfy some basic goals, it also incurs incidental complexity: boilerplate `visit()` methods in every concrete class, defining navigation visitors, and so forth. Leveraging Clojure's functional programming tools to implement the Visitor pattern is one way to get around this incidental complexity, while still programming on the JVM.

Trees in Clojure

Clojure provides a core set of persistent, immutable data structures: lists, vectors, maps, sets, and queues. Note that *persistent* here refers to a property of data modification, and not to data storage.

When a persistent data structure is "modified," the data is not mutated. Rather, a new immutable version of the data structure is returned. *Structural sharing* is the technique used to make Clojure's persistent data structures efficient. Structural sharing relies on the data's immutability to make sharing possible between the old and new instances of the data.

Immutability also allows us to simplify reasoning about concurrency and provide cheap reads of shared data. If the data is immutable, it can be read without acquiring a lock. If some other thread "changes" the data, the result will be a different instance of the data, which might share much of the same internal structure. In a functional programming language, persistent data structures are also important for writing pure functions, free of side-effects.

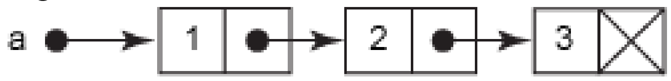
Clojure's persistent data structures

Making a basic linked-list data structure persistent is the easiest exercise; here, a list in Clojure is bound to a var named "a":

```
(def a (list 1 2 3))
```

Clojure is a Lisp variant, so evaluation occurs by first reading an expression as a Clojure data structure (usually a list), then evaluating each expression in the data structure. Finally, we invoke the first item in the list as a function with the rest of the items as arguments. The special form "`def`" will create a namespaced variable identified by the first symbol where the value is the expression following it. Each value in this linked list is held in a cell that contains a value and a pointer to the next cell (or *nil* to mark the end of the list), as shown in Figure 5:

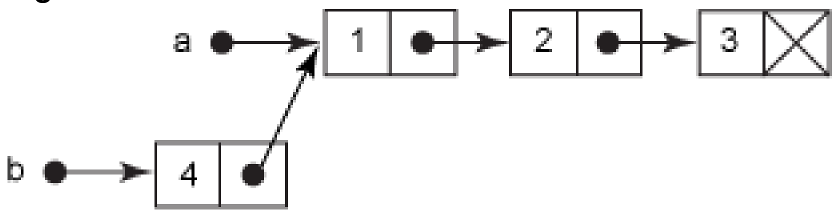
Figure 5. Linked list



If we then add a new value to the head of the list, it's known as constructing (or "cons-ing") a new list. The new list will share all the cells from the original:

```
(def b (cons 4 a))
```

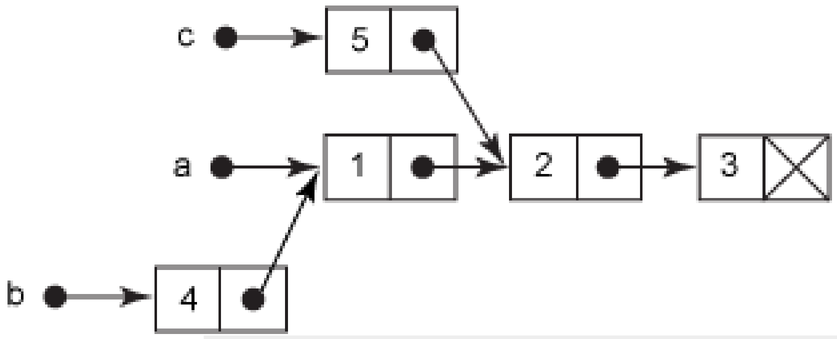
Figure 6. Linked list cons



We can use `rest` or other functions to access other parts of the list, but new lists created from the original list still share the original structure. Two key functions for operating on lists (and other sequences of values) are `first` (which returns the first item) and `rest` (which returns a list of the rest of the items).

```
(def c (cons 5 (rest a)))
```

Figure 7. More items in a linked list



> **Functional data structures**
> Clojure's other persistent structures, like vectors and maps, are implemented with hash-array mapped tries, as described by Phil Bagwell in "Ideal Hash Trees" (see Resources). That work is beyond the scope of this article, but all of the core data structures in Clojure use this technique.

The key difference in how we use Clojure versus Java data structures is that we do not mutate them in place; instead we describe an update and receive a new reference to an immutable structure that reflects the changes. When considering a tree of nodes where each node is an immutable data structure, we must consider how to modify a node or nodes inside the tree without modifying the entire tree.

Tree nodes

Before we consider the question of tree manipulation, let's consider the data structure we'll use to define each node of the tree. We want a well-defined set of properties for each type of node. It's also helpful if each node of the tree has a type visible to Clojure, which can be used when choosing a function implementation at runtime.

When considering a set of keys and values, the obvious choice in Clojure is the map, which stores key-value pairs. The code sample in Listing 4 demonstrates how to create a map, add values to it, and get values from it:

Listing 4. A map in Clojure

```
user> (def alex {:name "Alex" :eyes "blue"})
#'user/alex
user> alex
{:name "Alex", :eyes "blue"}
user> (:name alex)
"Alex"
user> (assoc alex :married true)
{:married true, :name "Alex", :eyes "blue"}
user> alex
{:name "Alex", :eyes "blue"}
```

Each of the keys in the map is a Clojure keyword, which always evaluates to itself and is identical to the same keyword used anywhere else. It is idiomatic to use keywords as keys in a map. Keywords are also functions. When invoked with a map, keyword functions look themselves up in the map and return their own value.

Adding entries to a map is done with `assoc` ("associate"), and removal is done with `dissoc` ("dissociate"). If you print a map you'll see commas between map entries, but it's purely for readability; commas are treated as whitespace in Clojure. At the end of Listing 4, `assoc` will return and print a new map, but the original map will be unmodified. The two maps will structurally share much of the same data.

Typed maps

Listing 5 shows some helper functions used to create maps with a well-known `:type` key. This type will be used later to dispatch polymorphic behavior. The `node-type` function extracts the "type" of a node based on the `:type` key.

Listing 5. Implementing typed tree nodes

```
(ns zippers.domain
  (:require [clojure.set :as set]))

(defn- expected-keys? [map expected-key-set]
  (not (seq (set/difference (set (keys map)) expected-key-set))))

(defmacro defnode
  "Create a constructor function for a typed map and a well-known set of
   fields (which are validation checked). Constructor will be
     (defn new-&node-type> [field-map])."
  [node-type [& fields]]
  (let [constructor-name (symbol (str "new-" node-type))]
    `(defn ~constructor-name [nv-map#]
       {:pre [(map? nv-map#)
              (expected-keys? nv-map# ~(set (map keyword fields)))]}
       (assoc nv-map# :type (keyword '~node-type)))))

(defn node-type [ast-node] (:type ast-node))
```

You might be seeing lots of new and advanced features in Listing 5, but don't panic! This code is included for the advanced Lisp or Clojure programmer, and it isn't essential to understand every detail. The core of the listing is the `defnode` macro that takes a node type and a vector of field names, and creates a constructor function for creating maps of that type. When the constructor is called, the fields being passed are checked to determine whether they match the expected fields, and an error is thrown if not. One of the benefits of Clojure as a Lisp variant is that code is data (also known as *homoiconicity*). Macros exploit this fact by manipulating code as data before evaluating it.

Listing 6 implements the Clojure equivalent to the Java domain classes:

Listing 6. Clojure domain types zippers/core.clj

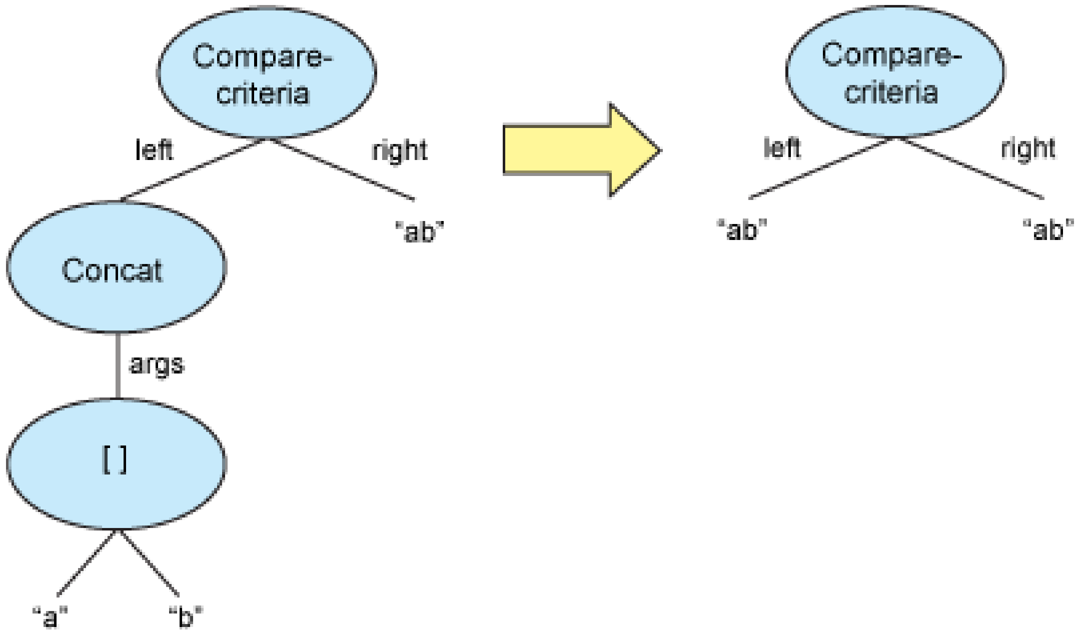
```
(defnode column [table column])
(defnode compare-criteria [left right])
(defnode concat [args])
(defnode filter [criteria child])
(defnode join [type left right])
(defnode project [projections child])
(defnode table [name])
(defnode value [value])
```

Walking the trees

Our challenge now is to take a tree of Clojure maps and traverse or manipulate it. Because Clojure trees are immutable data structures, any manipulation requires that we return a new, modified tree. As a first step, we can think back to our Java language implementation of the Visitor pattern and try something similar. We want to define an operation that walks over our tree of maps, searches for a pattern in the tree, and applies a manipulation at that point.

For example, we could search for `concat` functions with all string-literal values, then concatenate the arguments into a string value as in Figure 8:

Figure 8. Transforming a tree to evaluate the concat function



Similar to our Java Visitor solution, we could implement an `eval-concat` operation for every node type in the tree using a multimethod. In Clojure, a *multimethod* is a special kind of function that breaks invocation into two steps. When the function is invoked, a custom dispatch function is evaluated with the function's arguments. The resulting `dispatch-value` is then used to choose one of many possible function implementations.

```
(defmulti <function-name> <dispatch-function>)
(defmethod <function-name> <dispatch-value> [<function-args>] <body>)
```

A multimethod is defined by two macros: `defmulti` and `defmethod`. These macros are bound together by the `<function-name>`. The `defmulti` macro specifies the dispatch function to use when first invoked and a few other optional features. There will be many `defmethod` implementations, each specifying a `dispatch-value` that triggers execution, and a body of functions to execute when triggered.

Dispatching on type

The most common dispatch function in Clojure is simply the `class` function, such that invocation of the function implementation is based on the type of the sole function argument passed to the multimethod.

In our example, each node is a map and we want to use the `node-type` function to distinguish between the different types of maps representing each node. We then create implementations of the multimethod for each type we want to handle, using `defmethod`.

If the dispatch function does not determine any multimethod match possibilities, a `:default` dispatch value will be called, assuming it exists. In our example, we use a `:default` implementation to specify that all unlisted nodes should return themselves without modification. This serves as a useful base case, similar to a base `Visitor` class in the classic Visitor pattern.

Listing 7 is the complete implementation of the `eval-concat` multimethod. In this example, you see the use of functions like `new-concat` and `new-compare-criteria`, which were created by the `defnode` macro we called back in Listing 5.

Listing 7. Walking a tree with a multimethod

```
(defmulti eval-concat node-type)
(defmethod eval-concat :default [node] node)
(defmethod eval-concat :concat [concat]
           (let [arg-eval (map eval-concat (:args concat))]
             (if (every? string? arg-eval)
               (string/join arg-eval)
               (new-concat {:args arg-eval}))))
(defmethod eval-concat :compare-criteria [{:keys (left right) :as crit}]
           (new-compare-criteria {:left (eval-concat left)
                                  :right (eval-concat right)}))
```

Listing 8 is an example of the`eval-concat` multimethod in use. This example builds up a small tree of nodes, then executes the multimethod `eval-concat` on those nodes to find a `concat` of string literals. It then replaces them with the concatenated string.

Listing 8. Using the eval-concat multimethod

```
(def concat-ab (new-concat {:args ["a" "b"]}))
(def crit (new-compare-criteria {:left concat-ab
                                 :right "ab"}))

(eval-concat crit)
```

Note that the `eval-concat` in Listing 7 only partially solves our problem. For a full solution, we would need to create a `defmethod` for every class that might hold an expression, and thus a `concat` node. While not hard to do, it's tedious work.

Specifically, much of the "weight" of the solution involves traversing the data structure. All of the actual tree modification is isolated in the `:concat defmethod`. It would be better if traversing the data structure was a separate and generic operation.

Data traversal with clojure.walk

Clojure's core API includes a library, `clojure.walk`, that is specifically for traversing data structures. The library's functionality is based on a core function named `walk`, although `walk` is rarely directly called. Instead, it is far more common to access the functionality via the `prewalk` and `postwalk` functions. Both functions walk the tree in depth-first order but they differ in whether a node is visited before or after its children. Both versions take a function to apply at each node that returns a replacement node (or the original node).

For example, Listing 9 shows a `postwalk` operating on a heterogeneous tree of data. We first walk the tree and pass a function that merely prints the node being visited and returns the node. Then we call `postwalk`, passing in a function that looks for integers and increments them by one, leaving everything else the same.

Listing 9. The postwalk function at work

```
user> (def data [[1 :foo] [2 [3 [4 "abc"]] 5]])
#'user/data

user> (require ['clojure.walk :as 'walk])
nil

user> (walk/postwalk #(do (println "visiting:" %) %) data)
visiting: 1
visiting: :foo
visiting: [1 :foo]
visiting: 2
visiting: 3
visiting: 4
visiting: abc
visiting: [4 abc]
visiting: [3 [4 abc]]
visiting: 5
visiting: [2 [3 [4 abc]] 5]
visiting: [[1 :foo] [2 [3 [4 abc]] 5]]
[[1 :foo] [2 [3 [4 "abc"]] 5]]

user> (walk/postwalk #(if (number? %) (inc %) %) data)
[[2 :foo] [3 [4 [5 "abc"]] 6]]
```

Using `postwalk` and `prewalk` is beneficial because they already understand how to walk almost all of the core Clojure data structures — such as vectors, lists, sets, and maps (exceptions include sorted maps and records). When using `clojure.walk`, we don't need to specify any navigation code; instead we supply only the essential code that finds the node to modify and modifies it.

Let’s apply postwalk to our `eval-concat` problem from Listing 10. When we find a node of type `:concat`, we check whether it can be evaluated and return a new value node in place of the original `:concat` node.

Listing 10. The eval-concat problem with postwalk

```
(defn eval-concat [node]
  (if (and (= :concat (node-type node))
            (every? string? (:args node)))
```

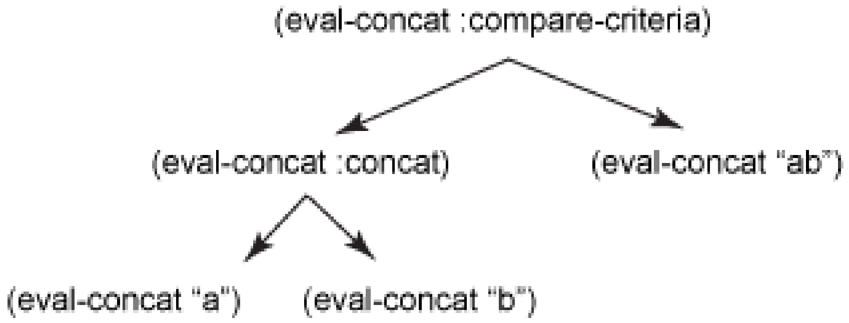
```
      (string/join (:args node))
      node))

(defn walk-example
  [node]
  (walk/postwalk eval-concat node))
```

That’s a much more satisfying implementation. All of the traversal is handled for us inside `postwalk` and we only need to supply the function to find and modify the tree in place.

Recursion in the Visitor pattern

One issue with both the original Visitor pattern and this implementation with `postwalk` is that the pattern is recursive (see Figure 9). When evaluating the modification function at a node, all of the parent-node traversals are on the stack. This is obvious in the Visitor implementation where you see all of the recursive calls to the `eval-concat` function. But even if it's tidily hidden away in `postwalk`, the structure is the same.

Figure 9. Recursive calls with clojure.walk



Recursion is not necessarily bad — it’s easy to understand and for small trees is unlikely to be an issue. But in the case of larger trees, we might prefer to traverse iteratively, in order to preserve memory. The question is, can we implement the Visitor pattern without recursion?

Clojure's functional zippers

A well-known solution to the problem of traversing and modifying a tree in functional programming is the zipper, most famously described by Gérard Huet (see Resources). A *zipper* is a way of representing a data structure with a local context, such that navigation (iteration) and modification from the current context are, for the most part, constant-time. All changes are local to the context.

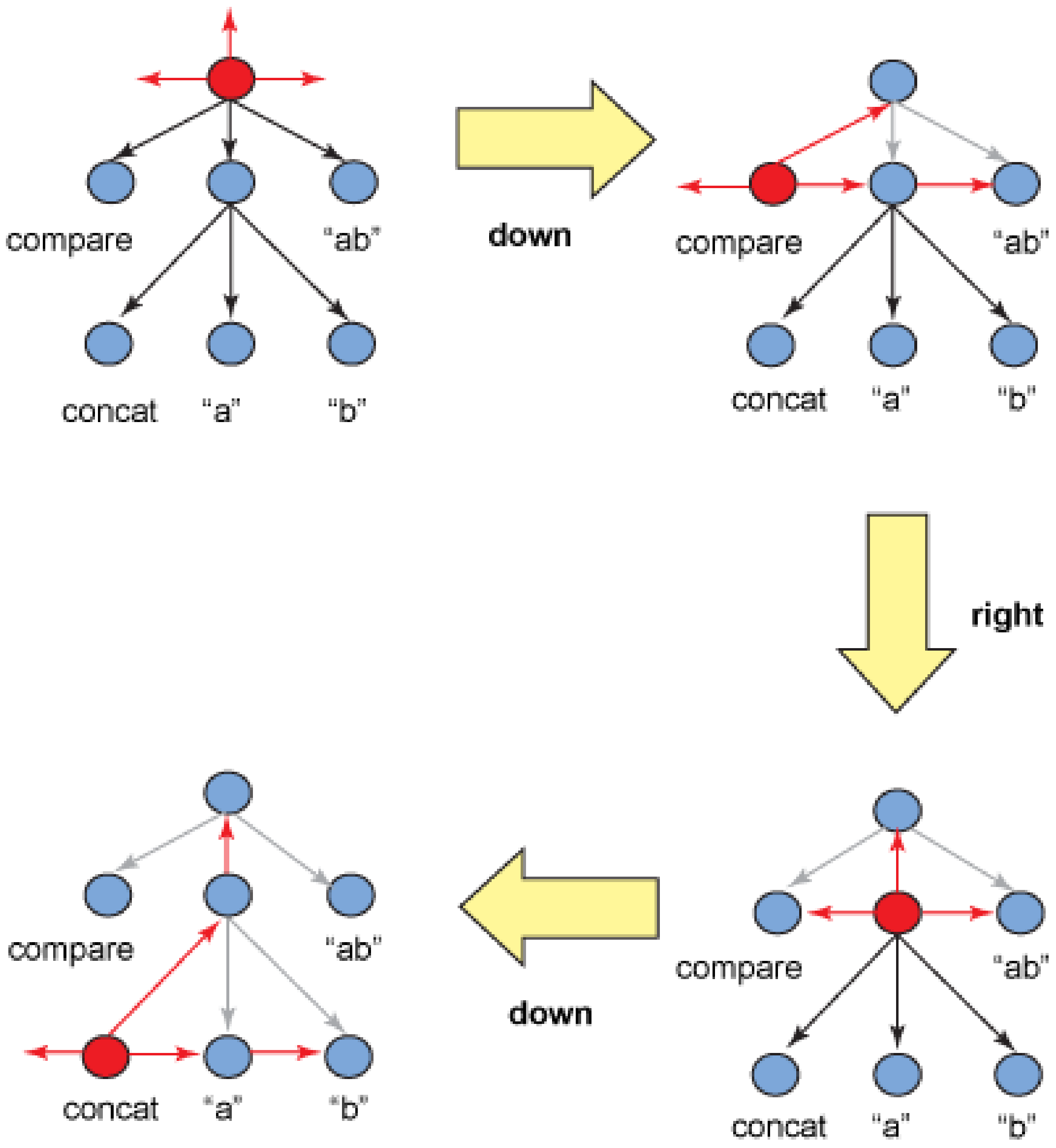
A *tree zipper* is typically composed of a path from the root to the current location in the tree, plus the subtree context at the focal node. The name "zipper" refers to moving up and down the tree like a zipper, with the part above the focal node as the open part of the zipper and the local state as the closed part.

In a typical tree, constant-time access is only available at the root (the node referenced by other code). Zippers allow that focal node to travel around the tree, getting constant-time access wherever you are (not including the traversal time to reach the node). A common way to think about zippers is that they are like a tree of rubber bands: you can "pick up" the tree at any node and it becomes like a root with the left, right, and parent paths hanging from that focal node.

The focal-node data structure is commonly referred to as a location, or "`loc`." The location contains the current subtree and the path to the root. Let’s consider a simplified version of the small tree structure from Figure 8. This time, we'll simplify the example by using vectors rather than maps.

Figure 10 illustrates how the location structure changes as we traverse the tree, here going down (always to the first left child), then right, then down. In each case, the new node becomes the focal point and the rest of the tree is represented as left nodes, right nodes, and parent nodes in relation to the focus.

Figure 10. Zipper loc structure while traversing the tree



As we traverse the tree, changes are local to the current focal node, meaning all constant-time operations except for up. That is the key benefit of zippers.

Zippers in Clojure

The Clojure core API contains an elegant zipper implementation in clojure.zip, with the API shown in Listing 11. I have divided the API functionality into several categories: construction, context, navigation, enumeration, and modification.

The *construction* functions allow you to create a new zipper (that is, a location). The core function for creating new zippers is `zipper`, which is based on three key functions:

- `branch?` takes a node and returns whether it's possible for that node to have children.
- `children` takes a node and returns the children of that node.
- `make-node` takes a node, a new set of children, and returns a new node instance.

The `seq-zip`, `vector-zip`, and `xml-zip` functions are helpers that call `zipper` with predefined implementations for sequences, vectors, and XML trees. (This implementation does not parse XML — it expects a data structure that represents XML, emitted by `clojure.xml/parse`.)

Listing 11. clojure.zip API

```
;; Construction
(zipper [branch? children make-node root]) - creates new zipper
(seq-zip [root]) - creates zipper made from nested seqs
(vector-zip [root]) - creates zipper made from nested vectors
(xml-zip [root]) - creates zipper from xml elements

;; Context
(node [loc]) - return node at current location
(branch? [loc]) - return whether the location is a branch in the tree
(children [loc]) - return the children of a location's node
(make-node [loc node children]) - make a new node for loc with the old node and the new
   children
(path [loc]) - return a seq of nodes leading to this location from the root
(lefts [loc]) - return a seq of nodes to the left
(rights [loc]) - return a seq of nodes to the right

;; Navigation
(left [loc]) - move to left sibling or return nil if none
(right [loc]) - move to right sibling or return nil if none
(leftmost [loc]) - move to leftmost sibling or self
(rightmost [loc]) - move to rightmost sibling or self
(down [loc]) - move to the leftmost child of the current location
(up [loc]) - move to the parent of the current location
(root [loc]) - move all the way to the root and return the root node

;; Enumeration
(next [loc]) - move to the next node in a depth-first walk
(prev [loc]) - move to the previous node in a depth-first walk
(end? [loc]) - at end of depth-first walk

;; Modification
(insert-left [loc item]) - insert a new left sibling
(insert-right [loc item]) - insert a new right sibling
(insert-child [loc item]) - insert a new leftmost child under current node
(append-child [loc item]) - inserts a new rightmost child under current node
(replace [loc node] - replaces the current node with a new node
(edit [loc edit-fn args]) - replace the current node with the results of edit-fn
(remove [loc]) - remove the current node and moves to the previous node in a depth-first
   walk
```

The *context* functions provide information about the current location in the tree and the *navigation* functions should be straightforward based on the previous discussion. Listing 12 shows an example of how to use the context and navigation functions to walk into and examine the tree:

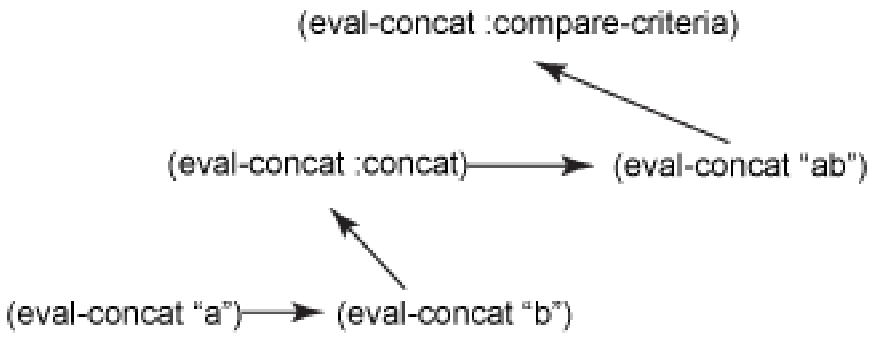
Listing 12. Traversing a tree of vectors with a zipper

```
> (def vz (zip/vector-zip [:compare [:concat "a" "b"] "ab"]))
> (println (zip/node (zip/down vz)))
:compare
> (println (zip/rights (zip/down vz)))
([:concat a b] ab)
> (println (zip/node (zip/right (zip/down vz))))
[:concat a b]
> (println (zip/node (zip/down (zip/right (zip/down vz)))))
:concat
```

Zipper Iteration

A zipper's *enumeration* functions allow you to traverse the entire tree in depth-first order, as shown in Figure 11. This traversal is interesting because it is iterative, not recursive, which is a key difference between zipper traversal and the prior `clojure.walk` implementation.

Figure 11. Iterative tree traversal with zippers



A tree visitor with zippers

Zipper enumeration and the zipper `edit` function give us the tools to build a zipper-based visitor, which consists of iterating the tree and executing a `visitor` function at each node. We can put these tools together as seen in Listing 13:

Listing 13. A zipper-based editor

```
(defn tree-edit [zipper matcher editor]
  (loop [loc zipper]
    (if (zip/end? loc)
      (zip/root loc)
      (if-let [matcher-result (matcher (zip/node loc))]
        (recur (zip/next (zip/edit loc (partial editor matcher-result))))
        (recur (zip/next loc))))))
```

The `tree-edit` function takes a `zipper` structure, a `matcher` function, and an `editor` function. This function starts with the Clojure special form `loop`, which creates a target for the `recur` at the end of the function. In Clojure, `loop/recur` indicates tail recursion and does not consume stack frames like other recursive calls. The `zip/next` call iterates to the next node in a depth-first walk through the tree.

The iteration terminates when `zip/end?` returns true. At that time, `zip/root` will `zip/up` to the top of the tree, applying any changes along the way, and return the root node. In the non-termination case, the `matcher` function is applied to the current `loc`. If it matches, the node and the result of the matcher are passed to the editor function for possible modification and iteration continues from the modified node. Otherwise, iteration continues from the original node. The `partial` function partially applies a function with a subset of its arguments and returns a new one that takes fewer arguments. In this case, we partially apply `editor` so that `edit` receives a new function with the appropriate signature.

We also need a zipper implementation that can deal with the standard Clojure data structures during traversal. The `tree-zipper` in Listing 14 implements the functions that enable the core collection types to allow any Clojure data structure built from those types to become a zipper: `branch?`, `children`, and `make-node`. I chose to use a multimethod for each zipper function, which allows me to dynamically extend this zipper later to other types, simply by adding new `defmethod` implementations.

Listing 14. Tree zipper implementation

```
(defmulti tree-branch? class)
(defmethod tree-branch? :default [_] false)
(defmethod tree-branch? IPersistentVector [v] true)
(defmethod tree-branch? IPersistentMap [m] true)
(defmethod tree-branch? IPersistentList [l] true)
(defmethod tree-branch? ISeq [s] true)

(defmulti tree-children class)
(defmethod tree-children IPersistentVector [v] v)
(defmethod tree-children IPersistentMap [m] (seq m))
(defmethod tree-children IPersistentList [l] l)
(defmethod tree-children ISeq [s] s)

(defmulti tree-make-node (fn [node children] (class node)))
(defmethod tree-make-node IPersistentVector [v children]
           (vec children))
(defmethod tree-make-node IPersistentMap [m children]
           (apply hash-map (apply concat children)))
(defmethod tree-make-node IPersistentList [_ children]
           children)
(defmethod tree-make-node ISeq [node children]
           (apply list children))
(prefer-method tree-make-node IPersistentList ISeq)

(defn tree-zipper [node]
  (zip/zipper tree-branch? tree-children tree-make-node node))
```

Concat evaluation

In Listing 15, we revisit our problem of evaluating a concat function by creating a match function (to find concat nodes with string-literal arguments) and an editor function (to evaluate the concat). The `can-simplify-concat` function acts as the matcher function and the `simplify-concat` function acts as the editor.

Listing 15. Concat evaluation with the zipper editor

```
(defn can-simplify-concat [node]
  (and (= :concat (node-type node))
       (every? string? (:args val))))

(defn simplify-concat [_ node]
  (string/join (:args node)))

(defn simplify-concat-zip [node]
  (tree-edit (tree-zipper node)
             can-simplify-concat
             simplify-concat))

(simplify-concat-zip crit)
```

So far, we have achieved almost the same level of essential complexity as with the clojure.walk implementation in Listing 10. One difference between Listing 10 and Listing 15 is that the walk version combined the "match" and "edit" parts, which are split in the zipper version. Another difference is that, internally, the zipper version uses a linear tail-recursive traversal of the data structure instead of a recursive traversal. This reduces memory usage during the iteration because the stack depth is constant instead of dependent on the height of the tree.

An even better tree visitor

Examining visitors that are useful in practice turns up several common categories based on the goal of the visitor:

- **Finder** searches for the first node matching some criteria and returns it.
- **Collector** searches for all nodes matching some criteria and returns them.
- **Transformer** searches for a match in the tree, mutates the tree at that point, and returns it.
- **Event generator** traverses the tree and fires events (think DOM to SAX).

Of course, you may find many other curious combinations and extensions of these categories as you begin to use and manipulate trees in practice. Our current `tree-edit` function does not allow the visitor to indicate that iteration should stop or to carry state through the traversal, so it is difficult to create Finder or Collector visitors without using external state holders.

Another observation after writing many visitors is that some common pieces of functionality should be reusable between visitors. For example, both a finder and a collector are looking to evaluate a criteria on a node and determine whether that node matches the criteria. Ideally, we would like to reuse a function that does this for both cases. Similarly, it is useful to create visitors that can check for cases that should cause skipping to the next node or aborting the iteration completely.

Listing 16 shows an enhanced visitor that applies a set of visitors at each node, passes state throughout the iteration, and allows for early exit of a node or the entire iteration. Note the two functions in use here:

- `tree-visitor` is similar to the `tree-edit` function previously discussed. It iterates through the tree via the zipper and terminates with `end?`. At each node, the `tree-visitor` calls `visit-node`, our second function. The primary difference in `tree-visitor` is that the `visit-node` function returns several items: a `new-node`, a `new-state`, and a `stop` flag. If `stop` is true, then iteration will exit immediately. The state is primed with `initial-state` and passed throughout the iteration, allowing the visitor functions to manipulate it in whatever way they desire. At the end of `tree-visitor`, both the final state and the final tree are returned.
- `visit-node` takes a list of visitor functions, each with a signature (`fn [node state]`) that returns a context map, which can contain the keys `:node`, `:state`, `:stop`, or `:jump`. If `:node` or `:state` are returned, they are replacements for the incoming node or state. Passing `:jump` indicates that iteration should jump to the next node and skip all remaining visitors for this node. Passing `:stop` indicates that all iteration should cease.

Listing 16. Enhanced tree visitor

```
(defn visit-node
  [start-node start-state visitors]
  (loop [node start-node
         state start-state
         [first-visitor & rest-visitors] visitors]
    (let [context (merge {:node node, :state state, :stop false, :next false}
                         (first-visitor node state))
          {new-node :node
           new-state :state
           :keys (stop next)} context]
      (if (or next stop (nil? rest-visitors))
        {:node new-node, :state new-state, :stop stop}
        (recur new-node new-state rest-visitors)))))

(defn tree-visitor
  ([zipper visitors]
     (tree-visitor zipper nil visitors))
  ([zipper initial-state visitors]
     (loop [loc zipper
            state initial-state]
       (let [context (visit-node (zip/node loc) state visitors)
             new-node (:node context)
```

```
              new-state (:state context)
              stop (:stop context)
              new-loc (if (= new-node (zip/node loc))
                        loc
                        (zip/replace loc new-node))
              next-loc (zip/next new-loc)]
          (if (or (zip/end? next-loc) (= stop true))
            {:node (zip/root new-loc) :state new-state}
            (recur next-loc new-state))))))
```

Using enhanced visitor functions

So what can we do with our new and improved visitor? Listing 17 shows an example collector that looks for strings in our tree. The `string-visitor` function looks for a string node and returns an updated state that captures the node. The `string-finder` calls the `tree-visitor` with `string-visitor` and returns the final state.

Listing 17. String finder

```
(defn string-visitor
  [node state]
  (when (string? node)
    {:state (conj state node)}))

(defn string-finder [node]
  (:state
   (tree-visitor (tree-zipper node) #{} [string-visitor])))
```

We can easily make a finder, too — let's make one that finds the first node of a certain type in Listing 18. The matched function is like our prior visitor functions but takes a node type to search for at the beginning. When `find-first` calls the visitor, it partially applies the type into `matched`, yielding a function that just takes `node` and `state` as expected. Note that the matched function returns `stop=true` to exit the iteration.

Listing 18. Node finder

```
(defn matched [type node state]
  (when (of-type node type)
    {:stop true
     :state node}))

(defn find-first [node type]
  (:state
   (tree-visitor (tree-zipper node) [(partial matched type)])))
```

Passing multiple functions

So far, we haven't actually leveraged the ability to pass multiple functions. The key here is that we want to decompose the functionality in our visitors into smaller pieces and then recombine them to build composite functionality. For example, in Listing

, we rewrote our `concat` evaluator to look for `:concat` nodes (function 1) that have all strings (function 2) and then evaluate the `concat` (function 3).

The first type evaluator visitor is generated in the generic "on" function. This function returns an anonymous visitor function that will jump to the next node if the node is not of the correct type. This function is completely reusable by any chain of visitors that needs to optionally evaluate the rest of the chain based on type. The second `all-strings` function similarly generates a conditional visitor that looks for a `concat` node with all string args.

Finally, we create a multimethod to handle the evaluation, called `eval-expr`. In this case, we default to evaluating anything by just returning itself. We add additional implementations for `:concat` and `:compare-criteria`. This multimethod is turned into a visitor with the `node-eval` function.

Assembling this chain of visitors into a composite visitor is then easy in `chained-example`, shown in Listing 19:

Listing 19. Using multiple small visitors

```
(defn on [type]
  (fn [node state]
    (when-not (of-type node type)
      {:jump true})))

(defn all-strings []
  (fn [{args :args} _]
    (when-not (every? string? args)
      {:jump true})))

(defmulti eval-expr node-type)
(defmethod eval-expr :default [x] x)
(defmethod eval-expr :concat [{args :args :as node}]
           (string/join args))
(defmethod eval-expr :compare-criteria [{:keys (left right) :as node}]
           (if (= left right) true node))

(defn node-eval [node state]
  {:node (eval-expr node)})

(defn chained-example [node]
  (:node
   (tree-visitor (tree-zipper node)
                 [(on :concat)
                  (all-strings)
                  node-eval])))
```

It’s easy to reuse several pieces of this implementation (like the on and `node-eval` functions) in other visitor chains as well. The notion of a visitor chain provides a very flexible framework with many options for how you structure and reuse the visitors on the tree.

Doing more with Clojure visitors

This article has just scratched the surface of using zippers in the Visitor pattern; the point of it is to whet your appetite for further exploration. For example, you might have noticed that the structures of `on` and `all-strings` in Listing 18 look similar. Both are functions that wrap a filter into a visitor. Instead of chaining functions that contain filters, we could wrap a filter visitor around conditions composed using normal Clojure connectors. The addition of custom macros to create and combine those conditions could make the code more composable.

In fact, pattern-matching libraries (like the new Clojure core.match library; see Resources) let you specify patterns with wildcards that should match in a data structure. It is not hard to integrate a pattern-matching library with the tree visitor to get to a point where visitors can leverage tree patterns. This is a powerful step closer to talking about your problem in the terms of your problem, letting the language itself get out of the way.

Another aspect of the Visitor pattern not closely examined here is iteration order. At Revelytix, we always traverse the nodes in the depth-first order of the zipper enumeration. In some cases, we may actually want to traverse these nodes in the reverse order, skip certain nodes or subtrees, or something else. Iteration essentially consists of three operations: `start` (find the `start loc` from the root node); `next` (given a `loc`, find the next `loc`), and `end?` (is the current `loc` the end of the iteration?). It is easy to abstract these functions in our current tree-visitor function and allow for pre-built and custom iteration strategies.

In Clojure, all code is also data that happens to be stored as trees of *s-expressions*. You can therefore use all of the techniques described in this article not just to modify your data, but to modify your code. You can find some examples of this in the core Clojure API's more advanced macro utilities.

I hope that you can use the ideas presented here to build your own visitors in Clojure or your language of choice, and to continue exploring ways to structure and manipulate your data. See the Resources section for a link to download all of the code and examples used in this article, which are stored on Github.

Acknowledgements

Many thanks go out to my colleagues at Revelytix who developed code similar to what's in this article for use in our own products. In particular, I am indebted to David McNeil for many fine hours talking through ideas and solutions on a whiteboard or pairing in Emacs, as well as for doing the heavy lifting to extend libraries like clojure.walk, clojure.zip, and matchure.

I am also thankful to my colleague Alex Hall for seeing the need and doing the implementation for the post-order iteration enhancement.

Resources

Learn

- Learn more about Clojure on developerWorks:
    - "The Clojure programming language" (Michael Galpin, 2009)
    - "Clojure and concurrency" (Michael Galpin, 2010)
    - "Solving the Expression Problem with Clojure 1.2" (Stuart Sierra, 2010)
    - "Using CouchDB with Clojure" (Ryan Senior, 2011)
- See Wikipedia's entry on zippers and Haskell's documentation for an overview of zippers.
- Gérard Huet introduced the concept of zippers in the article "Functional Pearl: The Zipper" (*Journal of Functional Programming, Volume 7*, Cambridge University Press, 1997).
- For more about Clojure zippers, see Luke VanderHart's video presentation on Blip.tv (2010).
- Brian Marick's "'Editing' trees in Clojure with clojure.zip" (Exploration Through Example, 2010) is a tutorial introduction to clojure.zip.
- See "Pattern Matching & Predicate Dispatch" (David Nolen, NYC Clojure Users Group, 2011) for an introduction to core.match.
- Philip Wadler, a principal designer of Haskell, defined the expression problem in 1998, while a researcher at Bell Labs.
- "Ideal Hash Trees" (Philip Bagwell, *Es Grands Champs*, Volume 1195, 2001) describes the data structure at the heart of Clojure's persistent data structures.

Get products and technologies

- Download the source code and examples from this article.
- Download core.match: An optimized pattern-matching and predicate-dispatch library for Clojure.

About the author

Alex Miller

Alex Miller is a senior engineer with Revelytix, building federated semantic web query technology. He has been working with Clojure full-time for two years. Prior to Revelytix, Alex was technical lead at

Terracotta, an engineer at BEA Systems, and chief architect at MetaMatrix. His interests include Java, concurrency, distributed systems, languages, and software design. Alex enjoys tweeting as @puredanger and blogging at http://tech.puredanger.com. Alex is the founder of the Strange Loop developer conference and the Lambda Lounge group for the study of functional and dynamic languages. He likes nachos.