```
#+BEGIN_SRC elisp
  (require 'ps-print)
  ps-print-buffer
  (setq ps-printer-name "LP1")
#+END_SRC


* Alt iterator

** Process

1. Write out the test code.
   a. Note that it's unspecified when iterators have different counts
2. Talk about the expected output
3. Work on implementation.
   a. Ask if anyone has any ideas.
   b. If no one does mine, do mine at the end.

** Initial code

#+BEGIN_SRC java
  public class Alt<X> implements Iterator<X>, Iterable<X> {
      Iterator<X> first;
      Iterator<X> second;

      Alt(Iterator<X> first, Iterator<X> second) {
          this.first = first;
          this.second = second;
      }

      public boolean hasNext() {
          return false;
      }

      public X next() {
          return null;
      }
  }
#+END_SRC

** Test code

#+BEGIN_SRC java
  Iterator<Integer> l123 = AListof.make(3, n -> n+1).iterator();
  Iterator<Integer> l456 = AListof.make(3, n -> n+4).iterator();
  Iterable<Integer> alt = new Alt<>(l123, l456);
  for (Integer i : alt) {
    System.out.println(i);
  }
  // Should print 1 4 2 5 3 6
#+END_SRC
```

```
** Finished code

#+BEGIN_SRC java
  public class Alt<X> implements Iterator<X>, Iterable<X> {
      Iterator<X> first;
      Iterator<X> second;

      Alt(Iterator<X> first, Iterator<X> second) {
          this.first = first;
          this.second = second;
      }

      public boolean hasNext() {
          return first.hasNext();
      }

      public X next() {
          X x = first.next();
          Iterator<X> tmp = this.first;
          this.first = this.second;
          this.second = tmp;
          return x;
      }

      public Iterator<X> iterator() {
          return this;
      }
  }
#+END_SRC
```

```
* List zipper

** Process

1. Try to call =l123.next()= after Alt test code.

2. Ask class why this doesn’t work.

3. Lament mutation, talk about how we lost the list.

4. Look at the details of =ListIterator=.

5. Copy into =Zipper.java=, rename to =ListZipper=,
   delete current =Iterator=, =Iterable= impl.

6. Motivate =context=, holding on to all the elements
   that we get rid of: everything we need to reconstruct
   the original list.

7. Implement =right=.
   #+BEGIN_SRC java
     ListZipper<X> right() {
       return this.list.accept(new ListVisitor<X, ListZipper<X>>() {
           public ListZipper<X> visitEmpty(Empty<X> mt) {
             return ListZipper.this;
           }
           public ListZipper<X> visitCons(Cons<X> cons) {
             return new ListZipper<>(new Cons<>(cons.first,
                     ListZipper.this.context), cons.rest);
           }
         });
     }
   #+END_SRC
```

```
8. Implement =Iterator=, =Iterable= as mutable =right=.
   #+BEGIN_SRC java
     public boolean hasNext() {
       return this.list.accept(new ListVisitor<X, Boolean>() {
           public Boolean visitEmpty(Empty<X> mt) {
             return false;
           }
           public Boolean visitCons(Cons<X> cons) {
             return true;
           }
         });
     }

     public X next() {
       return this.list.accept(new ListVisitor<X, X>() {
           public X visitEmpty(Empty<X> mt) {
             throw new RuntimeException("At the end of the list!");
           }
           public X visitCons(Cons<X> cons) {
             ListZipper<X> next = ListZipper.this.right();
             ListZipper.this.context = next.context;
             ListZipper.this.list = next.list;
             return cons.first;
           }
         });
     }

     public Iterator<X> iterator() {
       return this;
     }
   #+END_SRC

9. Show =Zip= test, and =Rezip= with nothing.
#+BEGIN_SRC java
  ListZipper<Integer> zip = new ListZipper(AListof.make(3, n -> n+1));

  System.out.println("Zip:");
  for (Integer i : zip) {
    System.out.println(i);
    System.out.println(zip);
    // Prints 1 2 3
  }

  System.out.println("Rezip:");
  for (Integer i : zip) {
    System.out.println(i);
    // Doesn’t print anything!
  }
#+END_SRC
```

```
10. Implement =left=.
#+BEGIN_SRC java
  ListZipper<X> left() {
      return this.context.accept(new ListVisitor<>() {
          public ListZipper<X> visitEmpty(Empty<X> mt) {
              return ListZipper.this;
          }
          public ListZipper<X> visitCons(Cons<X> cons) {
              return new ListZipper<>(cons.rest,
                      new Cons<>(cons.first, ListZipper.this.list));
          }
      });
  }
#+END_SRC

11. Show =Left= test.
#+BEGIN_SRC java
  System.out.println("Left:");
  for (Integer i : zip.left().left().left()) {
    System.out.println(i);
    // Prints 1 2 3
  }
#+END_SRC

12. Implement =start=, =end=.
#+BEGIN_SRC java
  ListZipper<X> start() {
      return this.context.accept(new ListVisitor<>() {
          public ListZipper<X> visitEmpty(Empty<X> mt) {
              return ListZipper.this;
          }
          public ListZipper<X> visitCons(Cons<X> cons) {
              return ListZipper.this.left().start();
          }
      });
  }

  ListZipper<X> end() {
      return this.list.accept(new ListVisitor<>() {
          public ListZipper<X> visitEmpty(Empty<X> mt) {
              return ListZipper.this;
          }
          public ListZipper<X> visitCons(Cons<X> cons) {
              return ListZipper.this.right().end();
          }
      });
  }
#+END_SRC

13. Show =Start= test.
#+BEGIN_SRC java
  System.out.println("Start:");
  for (Integer i : zip.start()) {
    System.out.println(i);
    // Prints 1 2 3
  }
#+END_SRC

14. Implement =flip=, ask what this does to the list.
#+BEGIN_SRC java
  ListZipper<X> flip() {
      return new ListZipper<>(this.list, this.context);
  }
#+END_SRC

15. Show =Flipped= test (after =Start=).
#+BEGIN_SRC java
  System.out.println("Flipped:");
  for (Integer i : zip.flip()) {
    System.out.println(i);
    // Prints 3 2 1
  }
```

```
#+END_SRC

16. Discuss benefits of functional iteration:
    a. No information is lost.
    b. We can move freely inside the list.
    c. We can reiterate at will.
    d. We can easily share the iterator.

** Test code

#+BEGIN_SRC java
  // After Alt test code (should throw runtime exception):
  System.out.println(l123.next());

  ListZipper<Integer> zip = new ListZipper(AListof.make(3, n -> n+1));

  System.out.println("Zip:");
  for (Integer i : zip) {
    System.out.println(i);
    System.out.println(zip);
    // Prints 1 2 3
  }

  System.out.println("Rezip:");
  for (Integer i : zip) {
    System.out.println(i);
    // Doesn't print anything!
  }

  System.out.println("Left:");
  for (Integer i : zip.left().left().left()) {
    System.out.println(i);
    // Prints 1 2 3
  }

  System.out.println("Start:");
  for (Integer i : zip.start()) {
    System.out.println(i);
    // Prints 1 2 3
  }

  System.out.println("Flipped:");
  for (Integer i : zip.flip()) {
    System.out.println(i);
    // Prints 3 2 1
  }
#+END_SRC

** Finished code

#+BEGIN_SRC java
  class ListZipper<X> implements Iterator<X>, Iterable<X> {

      // How to reconstruct the list
      Listof<X> context;
      Listof<X> list;

      ListZipper(Listof<X> l) {
          this(AListof.empty(), l);
      }

      ListZipper(Listof<X> context, Listof<X> list) {
          this.context = context;
          this.list = list;
      }

      ListZipper<X> right() {
          return this.list.accept(new ListVisitor<X, ListZipper<X>>() {
              public ListZipper<X> visitEmpty(Empty<X> mt) {
                  return ListZipper.this;
              }
              public ListZipper<X> visitCons(Cons<X> cons) {
                  return new ListZipper<>(new Cons<>(cons.first, ListZipper.this.context), cons.r
est);
              }
          });
      }

      ListZipper<X> left() {
          return this.context.accept(new ListVisitor<>() {
              public ListZipper<X> visitEmpty(Empty<X> mt) {
                  return ListZipper.this;
              }
              public ListZipper<X> visitCons(Cons<X> cons) {
                  return new ListZipper<>(cons.rest, new Cons<>(cons.first, ListZipper.this.list)
);
              }
          });
      }

      ListZipper<X> start() {
          return this.context.accept(new ListVisitor<>() {
              public ListZipper<X> visitEmpty(Empty<X> mt) {
                  return ListZipper.this;
              }
              public ListZipper<X> visitCons(Cons<X> cons) {
                  return ListZipper.this.left().start();
              }
          });
      }

      ListZipper<X> end() {
          return this.list.accept(new ListVisitor<>() {
              public ListZipper<X> visitEmpty(Empty<X> mt) {
                  return ListZipper.this;
              }
              public ListZipper<X> visitCons(Cons<X> cons) {
                  return ListZipper.this.right().end();
              }
          });
      }

      ListZipper<X> flip() {
          return new ListZipper<>(this.list, this.context);
      }

      Listof<X> unzip() {
          return this.end().list;
      }

      public boolean hasNext() {
          return this.list.accept(new ListVisitor<X, Boolean>() {
              public Boolean visitEmpty(Empty<X> mt) {
                  return false;
              }
              public Boolean visitCons(Cons<X> cons) {
                  return true;
              }
          });
      }

      public X next() {
          return this.list.accept(new ListVisitor<X, X>() {
              public X visitEmpty(Empty<X> mt) {
                  throw new RuntimeException("At the end of the list!");
              }
              public X visitCons(Cons<X> cons) {
                  ListZipper<X> next = ListZipper.this.right();
                  ListZipper.this.context = next.context;
                  ListZipper.this.list = next.list;
                  return cons.first;
              }
          });
      }
```

```
      public Iterator<X> iterator() {
          return this;
      }

      public String toString() {
          return "?" + context + ", " + list + "?";
      }
  }
#+END_SRC

* Tree

#+BEGIN_SRC java
  @FunctionalInterface
  interface TriFunction<T, U, V, R> {
      // Apply this ternary function
      R apply(T t, U u, V v);
  }

  interface TreeVisitor<X, R> {
      R visitLeaf(Leaf<X> leaf);
      R visitNode(Node<X> node);
  }

  interface Tree<X> {
      <R> R fold(TriFunction<X, R, R, R> f, R b);
      <R> R accept(TreeVisitor<X, R> visitor);
  }

  class Leaf<X> implements Tree<X> {
      public <R> R fold(TriFunction<X, R, R, R> f, R b) {
          return b;
      }

      public <R> R accept(TreeVisitor<X, R> visitor) {
          return visitor.visitLeaf(this);
      }
  }

  class Node<X> implements Tree<X> {
      X value;
      Tree<X> left;
      Tree<X> right;

      Node(X value, Tree<X> left, Tree<X> right) {
          this.value = value;
          this.left = left;
          this.right = right;
      }

      public <R> R fold(TriFunction<X, R, R, R> f, R b) {
          return f.apply(
                  this.value,
                  this.left.fold(f, b),
                  this.right.fold(f, b));
      }

      public <R> R accept(TreeVisitor<X, R> visitor) {
          return visitor.visitNode(this);
      }
  }
#+END_SRC

* TreeZipper

#+BEGIN_SRC java
  class TreeZipper<X> {

      interface Pieces<X> {
          Tree<X> remake(Tree<X> tree);
      }
```

```
    class Right<X> implements Pieces<X> {
        X value;
        Tree<X> right;
        Right(X value, Tree<X> right) {
            this.value = value;
            this.right = right;
        }
        public Tree<X> remake(Tree<X> left) {
            return new Node<X>(value, left, right);
        }
    }

    class Left<X> implements Pieces<X> {
        X value;
        Tree<X> left;
        Left(X value, Tree<X> left) {
            this.value = value;
            this.left = left;
        }
        public Tree<X> remake(Tree<X> right) {
            return new Node<X>(value, left, right);
        }
    }

    // How to reconstruct the tree
    Listof<Pieces<X>> context;
    Tree<X> current;

    TreeZipper(Tree<X> tree) {
        this(AListof.empty(), tree);
    }

    TreeZipper(Listof<Pieces<X>> context, Tree<X> tree) {
        this.context = context;
        this.current = tree;
    }

    TreeZipper<X> right() {
        return this.current.accept(new TreeVisitor<X, TreeZipper<X>>() {
            public TreeZipper<X> visitLeaf(Leaf<X> leaf) {
                return TreeZipper.this;
            }

            public TreeZipper<X> visitNode(Node<X> node) {
                return new TreeZipper<>(
                        context.cons(new Left<>(node.value, node.left)),
                        node.right
                );
            }
        });
    }

    TreeZipper<X> left() {
        return this.current.accept(new TreeVisitor<X, TreeZipper<X>>() {
            public TreeZipper<X> visitLeaf(Leaf<X> leaf) {
                return TreeZipper.this;
            }

            public TreeZipper<X> visitNode(Node<X> node) {
                return new TreeZipper<>(
                        context.cons(new Right<>(node.value, node.right)),
                        node.left
                );
            }
        });
    }

    TreeZipper<X> up() {
        return this.context.accept(new ListVisitor<Pieces<X>, TreeZipper<X>>() {
            public TreeZipper<X> visitEmpty(Empty<Pieces<X>> mt) {
                return TreeZipper.this;
```

```
              }

              public TreeZipper<X> visitCons(Cons<Pieces<X>> cons) {
                  Pieces<X> pieces = cons.first;
                  return new TreeZipper<>(
                          cons.rest,
                          pieces.remake(TreeZipper.this.current)
                  );
              }
          });
      }
  }
#+END_SRC
```