# Challenge 1

## List Reversal

### Background

In-place reversal of a singly-linked list is a classical algorithm. It is efficient (constant space and linear time) and straightforward to implement in any mainstream programming language. Here is a C implementation:

```C/C++
struct Cell { int value; struct Cell *next; };

struct Cell *list_reversal(struct Cell *l) {
  struct Cell *r = NULL;
  while (l != NULL) {
    struct Cell *tmp = l;
    l = l->next;
    tmp->next = r;
    r = tmp;
  }
  return r;
}
```

List reversal has been widely studied, notably in the context of program verification. In particular, it is the archetypal case study in Separation Logic, and the very first example in Reynolds's landmark paper [LICS 2002].

It is clear that the code above successfully terminates on a NULL-terminated list. Perhaps less well-known is the fact that it also successfully terminates on any infinite list that loops at some point, i.e., a lasso-shaped list with an initial segment (of any length $\geq 0$) and then a cycle (of any length $\geq 1$). When applied to a list like this, list reversal reverses the cycle and leaves the initial segment unchanged (by reversing it twice).

If the memory is finite, any list is either NULL-terminated or loops at some point, and thus list reversal always terminates.

### Tasks

1. Implement list reversal in a language of your choice.
2. Show that your implementation, if invoked on a well-formed list, is free of runtime violations; in particular, no invalid pointer operation occurs. Part of this task is to specify precisely "well-formed." (Both NULL-terminated and lasso-shaped lists are considered well-formed.)
3. Show that your implementation, if invoked on any well-formed list in finite memory, terminates.
4. Show that, at termination, the resulting list is the reverse of the original. For lasso-shaped lists, this means the initial segment is unchanged and the cycle is reversed, as explained above.
5. Show that the space and time consumed are linear in the length of the list.

*Thanks to Jean-Christophe Filliâtre and Andrei Paskevich for contributing this problem.*