This error occurs when an attempt is made to borrow state past the end of the lifetime of a type that implements the `Drop` trait.

Erroneous code example:

```
#![feature(nll)]

pub struct S<'a> { data: &'a mut String }

impl<'a> Drop for S<'a> {
    fn drop(&mut self) { self.data.push_str("being dropped"); }
}

fn demo<'a>(s: S<'a>) -> &'a mut String { let p = &mut *s.data; p }
```

Here, `demo` tries to borrow the string data held within its argument `s` and then return that borrow. However, `S` is declared as implementing `Drop`.

Structs implementing the `Drop` trait have an implicit destructor that gets called when they go out of scope. This destructor gets exclusive access to the fields of the struct when it runs.

This means that when `s` reaches the end of `demo`, its destructor gets exclusive access to its `&mut`-borrowed string data. allowing another borrow of that string data (`p`), to exist across the drop of `s` would be a violation of the principle that `&mut`-borrows have exclusive, unaliased access to their referenced data.

This error can be fixed by changing `demo` so that the destructor does not run while the string-data is borrowed; for example by taking `S` by reference:

```
pub struct S<'a> { data: &'a mut String }

impl<'a> Drop for S<'a> {
    fn drop(&mut self) { self.data.push_str("being dropped"); }
}

fn demo<'a>(s: &'a mut S<'a>) -> &'a mut String { let p = &mut *(*s).data; p }
```

Note that this approach needs a reference to S with lifetime `'a`. Nothing shorter than `'a` will suffice: a shorter lifetime would imply that after `demo` finishes executing, something else (such as the destructor!) could access `s.data` after the end of that shorter lifetime, which would again violate the `&mut`-borrow's exclusive access.