COMP 333
Fall 2024

Heap Allocation and Generics in Rust (Answers)

1.a.) Define an `enum` named `IntList` which encodes a singly-linked list of 32-bit signed integers, using the same cons/nil structure that we used in assignment 1. For the cons case, the rest of the list will need to be heap-allocated via `Box`.

```
enum IntList {
  Nil,
  Cons(i32, Box<IntList>)
}
```

1.b.) Using the definition from 1.a., create a list containing 1, 2, and 3, in that order.

```
IntList::Cons(1, Box::new(
  IntList::Cons(2, Box::new(
    IntList::Cons(3, Box::new(
      IntList::Nil))))))
```

1.c.) Using the definition from 1.a., write a `length` method that will take a reference to an `IntList`, and will return the length of the `IntList`. The length should be represented with a 64-bit unsigned integer.

```
impl IntList {
  fn length(&self) -> u64 {
    match self {
      IntList::Nil => 0,
      IntList::Cons(_, tail) => 1 + tail.length()
    }
  }
}
```

2.a.) Define an `enum` named `List`, which has the same structure as `IntList` from 1.a., but will work with a generic type `A` instead of an integer.

```
enum List<A> {
    Nil,
    Cons(A, Box<List<A>>)
}
```

2.b.) Define a `prepend` method for `List`, which will take ownership over a `List` instance, as well as an element `e` of the same type that the list contains. `prepend` will return a `List` that starts with `e`, and is followed by the rest of the elements in the list. Example usage is below:

```
// creates the list [3, 2, 1]
let list: List<i32> =
  List::Nil.prepend(1).prepend(2).prepend(3);

impl<A> List<A> {
    fn prepend(self, e: A) -> List<A> {
        List::Cons(e, Box::new(self))
    }
}
```

2.b.) Define a `head` method for `List`, which will return either a `Some` holding a reference to the first element of the `List` (for a `Cons`), or a `None` if the list is `Nil`. `head` should not need ownership over the List, only a reference to the List.

```
impl<A> List<A> {
    fn head(&self) -> Option<&A> {
        match self {
            List::Cons(head, _) => Some(&head),
            List::Nil => None
        }
    }
}
```