

Guile Activity

Use Guile to do the following big list of things.

Don't look up the answers or ask an LLM; this will rob you of the joy of finding the answer yourself.

Feel free to work in small groups.

Implement a function `inc`, which takes a single argument, and adds 1 to it.

e.g.,

```
(inc 1)
```

should evaluate to 2.

Implement a function `fact`, which takes an argument `n`, and returns `n` factorial.

e.g.,

```
(fact 5)
```

should evaluate to 120.

Implement a function `inc-list`, which takes a list `l` as an argument, and returns a list obtained by applying `inc` to each element of `l`.

e.g.,

```
(inc-list '(1 2 3 4))
```

should evaluate to (2 3 4 5).

Implement a function `range`, which takes an argument `n`, and returns the list containing the numbers 0 to `n`, exclusive on the right side.

e.g.,

```
(range 5)
```

should evaluate to (0 1 2 3 4).

Implement a function `map`, which takes a function `f` and a list `l` as arguments, and returns a list obtained by applying `f` to each element of `l`.

e.g.,

```
(map inc '(1 2 3 4))
```

should evaluate to (2 3 4 5).

Implement `inc-list` using `map`.

Implement a function `evens`, which takes a list `l`, and returns a list containing the even elements of `l`.

Feel free to use the built-in function `even?`.

e.g.,

```
(evens '(1 2 3 4 5 6))
```

should evaluate to (2 4 6).

Implement a function `filter`, which takes a function `f` and a list `l`, and returns a list of each value `x` in `l` such that `(f x)` evaluates to `#t`.

e.g.,

```
(filter even? '(1 2 3 4 5 6))
```

should evaluate to `(2 4 6)`.

Implement `evens` using `filter`.

Implement a function `append-back`, which takes a list `l` and a value `x`, and returns a new list obtained by appending `x` to `l`.

e.g.,

```
(append-back '(1 2 3 4) 5)
```

should evaluate to `(1 2 3 4 5)`.

Implement a function `rev`, which takes a list `l` as an argument, and returns a list obtained by reversing `l`.

e.g.,

```
(rev '(1 2 3 4))
```

should evaluate to `(4 3 2 1)`.

Implement a function `prod`, which takes a list `l` as an argument, and returns the product of the elements of `l`.

e.g.,

```
(prod '(5 2 3 3))
```

should evaluate to 90.

Implement a function `fold`, which takes 3 arguments:

1. `f`: a function of 2 arguments
2. `i`: an initial value
3. `l`: a list of values

and accumulates the values in `l` using `f`, starting with `i`.

That is,

```
(fold * 1 '(5 2 3 3))
```

should evaluate to

```
(* 3 (* 3 (* 2 (* 5 1))))
```

which should evaluate to 90.

Implement `prod` using `fold`.

Implement `fact` using `fold` and other previously-defined function(s).

Implement `rev` using `fold`.

Implement `map` using `fold` and other previously-defined function(s).