Lab 8: The Ambiguous Choice Operator

In the past two weeks of lecture, we've looked at an implementation of the *ambiguous choice operator* `-<`, and some applications of it. In this lab, you'll do some more work towards applying the choice operator to interesting problems.

Starter code

- `streams.rkt`
- `amb.rkt`
- `lab8.rkt`

Task 1: Making change

Let's look at one application of choice expressions to solve a famous problem in computer science: finding combinations of coins whose value equals a given amount.

Your first task here is to complete `make-change`, which yields combinations of 1's and 5's that sum to a given number. As you might expect, this is a choice function, meaning it returns one choice; the remaining choices are accessible by calling `next`:

```
> (make-change 10)
'(5 5)
> (next)
'(5 1 1 1 1 1)
> (next)
'(1 1 1 1 1 1 1 1 1 1)
> (next)
'done
```

Your next task is to generalize this to a function `make-change-gen` so that it takes two arguments: a list of coin values, and a target number.

Note: You might see strange behaviours when calling functions inside an `-<` expression, where the function also uses `-<` to generate choices. The workaround is to

1. First generate the choices (e.g. `-<` generates the arguments to the function that you would like to call)
2. Call the function.

The strange behaviour occurs when functions used in expressions inside `-<` uses `(fail)` to backtrack.

For example, if `f` uses `-<` and sometimes calls `(fail)`, you should *not* write:

```
(-< (f 3 5) (f 4 6))
```

Instead, choose the argument to `f` first, and then apply `f`:

```
(let* ([choice (-< '(3 5) '(4 6))])   (apply f choice))
```