Function Usage Patterns

We describe here a variety of usage patterns that involve functions and closures. We have already seen powerful usages in terms of creating new scopes. These are just more examples.

Relevant Links

- Flanagan's book, section 8.8.4

Notes

Memoize

The idea here is that we would effectively cache results of function calls. If the function is later called with the same arguments, we would return the cached result directly.

This assumes a couple of things:

- That the function is what we would call "pure": It would do the exact same thing given the same inputs. In particular it cannot have any side-effects like printing something or setting an external value to something. It should just do a computation.
- That the arguments can be "stringified". We will store them as keys in a "cache" object, and if they cannot be properly converted to strings we will have problems. For instance objects all get converted to the string `'[object Object]'`, so if our arguments are objects then we're in trouble, as all objects will be seen as the exact same argument.

Here is a simple version of this idea, which assumes only one argument:

```
function memoize(f) {
    var cache = {};
    return function(v) {
        if (!cache.hasOwnProperty(v)) { cache[v] = f(v); }
        return cache[v];
    };
}
```

You can expand on it to limit the cache size or to allow multiple arguments, but we will not do so here.

Call Once

In this example, we want to make sure a specific function is only called once, and any subsequent "calls" just return the same value. This mostly makes sense for functions that take no arguments.

This is often used for “lazy loading”, where you have some module that would require a lot of work to initialize, and you only want to do it if and when it is actually needed. But once it is initialized once, there is no need to initialize it a second time, so subsequent “initialization calls” can simply recall the stored value.

You can also imagine a variant that allows a specific number of calls before it defaults to just returning the last computed value.

```
function once(f) {
    var called = false;
    var value;
    return function() {
        if (!called) {
            value = f();
            called = true;
        }
        return value;
    };
}
```

Here is an interesting variation:

```
function once(f) {
    var wrapper = function() {
        var value = f();
        wrapper = function() { return value; };
        return value;
    };
    return function() { return wrapper(); };
}
```

Food for thought: What would happen if we just return wrapper, instead of a function calling wrapper?