```
let gridBreakpoints = {}

//
// Helper Functions
//

function lookupVariable(context, variableName) {
	const { frames, importantScope } = context

	return tree.Variable.prototype.find(frames, frame => {
		const { value, important } = frame.variable(variableName) || {}

		if (value === undefined)
			return

		if (important && importantScope[importantScope.length - 1])
			importantScope[importantScope.length - 1].important = important

		return value.eval(context)
	})
}

// @TODO: [@calvinjuarez] unify this function between files, maybe even canonize it as a
// `Ruleset`/`DetachedRuleset` method at some point.
function rulesetToMap(context, { ruleset: { rules } } = { ruleset: { rules: [] } }) {
	const map = {}

	rules.forEach(rule => {
		// Not exactly sure how to handle other types (or if they should be handled at all).
		if (! (rule instanceof tree.Declaration))
			return

		const { name: key, value } = rule.eval(context)

		map[key] = value
	})

	return map
}

function getBreakpoints(context, breakpoints) {
	if (! breakpoints) {
		if (Object.keys(gridBreakpoints).length === 0)
			gridBreakpoints = lookupVariable(context, '@grid-breakpoints')

		breakpoints = gridBreakpoints
	}

	const rulesetMap = rulesetToMap(context, breakpoints)

	// Since values in the map will be instances of `Anonymous`, convert them to `Dimension`s.
	for (const key in rulesetMap) {
		const value = rulesetMap[key].value
		const number = parseFloat(value, 10)
		const unit = value.toString().replace(number, '')

		rulesetMap[key] = new tree.Dimension(number, unit)
	}

	return rulesetMap
}

//
// Less Functions
//

functions.add('breakpoint-next', function ({ value: breakpointName }, breakpoints) {
	const breakpointsMap  = getBreakpoints(this.context, breakpoints)
	const breakpointNames = Object.keys(breakpointsMap)
	const breakpointIndex = breakpointNames.indexOf(breakpointName)
```

```
        if (breakpointIndex === -1)
                return new tree.Quoted('"')

        // Next breakpoint is null for the last breakpoint.
        if ((breakpointIndex + 1) === breakpointNames.length)
                return new tree.Quoted('"')

        return new tree.Quoted('"', breakpointNames[breakpointIndex + 1])
})

functions.add('breakpoint-min', function ({ value: breakpointName }, breakpoints) {
        const breakpointsMap  = getBreakpoints(this.context, breakpoints)
        const breakpointNames = Object.keys(breakpointsMap)
        const breakpointIndex = breakpointNames.indexOf(breakpointName)

        if (breakpointIndex === -1)
                return new tree.Quoted('"')

        // Minumum breakpoint width is null for the first breakpoint.
        if (breakpointIndex === 0)
                return new tree.Quoted('"')

        return breakpointsMap[breakpointName]
})

functions.add('breakpoint-max', function ({ value: breakpointName }, breakpoints) {
        const breakpointsMap  = getBreakpoints(this.context, breakpoints)
        const breakpointNames = Object.keys(breakpointsMap)
        const breakpointIndex = breakpointNames.indexOf(breakpointName)

        if (breakpointIndex === -1)
                return new tree.Quoted('"')

        // Maximum breakpoint width is null for the last breakpoint.
        if ((breakpointIndex + 1) === breakpointNames.length)
                return new tree.Quoted('"')

        const nextBreakpoint = breakpointsMap[breakpointNames[breakpointIndex + 1]]

        return new tree.Dimension(nextBreakpoint.value - 0.02, nextBreakpoint.unit)
})

functions.add('breakpoint-infix', function ({ value: breakpointName }, breakpoints) {
        const breakpointsMap  = getBreakpoints(this.context, breakpoints)
        const breakpointNames = Object.keys(breakpointsMap)
        const breakpointIndex = breakpointNames.indexOf(breakpointName)

        if (breakpointIndex === -1)
                return new tree.Quoted('"')

        // Breakpoint infix is null the first breakpoint.
        if (breakpointIndex === 0)
                return new tree.Quoted('"')

        return new tree.Quoted('"', `-${breakpointName}`)
})
```