scope.js

J. Ashkenas

05-04-2013

Contents

The **Scope** class regulates lexical scoping within CoffeeScript. As you generate code, you create a tree of scopes in the same shape as the nested function bodies. Each scope knows about the variables declared within it, and has a reference to its parent enclosing scope. In this way, we know which variables are new and need to be declared with `var`, and which are shared with external scopes.

Import the helpers we plan to use.

```
{extend, last} = require './helpers'

exports.Scope = class Scope
```

The `root` is the top-level **Scope** object for a given file.

```
  @root: null
```

Initialize a scope with its parent, for lookups up the chain, as well as a reference to the **Block** node it belongs to, which is where it should declare its variables, and a reference to the function that it belongs to.

```
  constructor: (@parent, @expressions, @method) ->
    @variables = [{name: 'arguments', type: 'arguments'}]
    @positions = {}
    Scope.root = this unless @parent
```

Adds a new variable or overrides an existing one.

```
  add: (name, type, immediate) ->
    return @parent.add name, type, immediate if @shared and not ⤸
     ⤹ immediate
    if Object::hasOwnProperty.call @positions, name
      @variables[@positions[name]].type = type
    else
      @positions[name] = @variables.push({name, type}) - 1
```

When `super` is called, we need to find the name of the current method we're in, so that we know how to invoke the same method of the parent class. This can get complicated if super is being called from an inner function. `namedMethod` will walk up the scope tree until it either finds the first function object that has a name filled in, or bottoms out.

```
  namedMethod: ->
    return @method if @method?.name or !@parent
    @parent.namedMethod()
```

Look up a variable name in lexical scope, and declare it if it does not already exist.

```
  find: (name) ->
    return yes if @check name
    @add name, 'var'
    no
```

Reserve a variable name as originating from a function parameter for this scope. No `var` required for internal references.

```
parameter: (name) ->
  return if @shared and @parent.check name, yes
  @add name, 'param'
```

Just check to see if a variable has already been declared, without reserving, walks up to the root scope.

```
check: (name) ->
  !!(@type(name) or @parent?.check(name))
```

Generate a temporary variable name at the given index.

```
temporary: (name, index) ->
  if name.length > 1
    '_' + name + if index > 1 then index - 1 else ''
  else
    '_' + (index + parseInt name, 36).toString(36).replace  ⤸
     ↪ /\d/g, 'a'
```

Gets the type of a variable.

```
type: (name) ->
  return v.type for v in @variables when v.name is name
  null
```

If we need to store an intermediate result, find an available name for a compiler-generated variable. `_var`, `_var2`, and so on…

```
freeVariable: (name, reserve=true) ->
  index = 0
  index++ while @check((temp = @temporary name, index))
  @add temp, 'var', yes if reserve
  temp
```

Ensure that an assignment is made at the top of this scope (or at the top-level scope, if requested).

```
assign: (name, value) ->
  @add name, {value, assigned: yes}, yes
  @hasAssignments = yes
```

Does this scope have any declared variables?

```
hasDeclarations: ->
  !!@declaredVariables().length
```

Return the list of variables first declared in this scope.

```
declaredVariables: ->
  realVars = []
  tempVars = []
  for v in @variables when v.type is 'var'
    (if v.name.charAt(0) is '_' then tempVars else  ⤸
     ↪ realVars).push v.name
  realVars.sort().concat tempVars.sort()
```

Return the list of assignments that are supposed to be made at the top of this scope.

```
assignedVariables: ->
  "#{v.name} = #{v.type.value}" for v in @variables when ⤸
    ⤹ v.type.assigned
```