Naming Anonymous JavaScript Functions

Salman Mirghasemi
École Polytechnique Fédérale de Lausanne(EPFL)
salman.mirghasemi@epfl.ch

John J. Barton
IBM Research - Almaden
bartonjj@us.ibm.com

Claude Petitpierre
École Polytechnique Fédérale de Lausanne(EPFL)
claude.petitpierre@epfl.ch

Abstract

JavaScript developers create programs by calling functions and they use functions to construct objects. JavaScript development tools need to report to developers about those functions and constructors, for example in debugger call-stacks and in object representations. However, most functions are anonymous: developers need not to specify names for functions. Based on our analysis of ten large, widely used JavaScript projects, less than 7% of JavaScript functions are named by developers. After studying examples from these JavaScript projects, we propose *Static Function-Object Consumption*, a principled, automated approach based on local source code analysis for providing names to nameless JavaScript functions. We applied our approach to 90000 anonymous functions that appeared in the analyzed JavaScript projects. The approach is successful in naming more than 99% (91% are unique within their file) of anonymous functions while the average length of function names is kept less than 37 characters.

Categories and Subject Descriptors D.2.5 [*Testing and Debugging*]: Debugging aids; D.2.6 [*Programming Environments*]: Integrated environments

General Terms Algorithms, Human Factors, Languages

Keywords JavaScript, Anonymous Function Name, Debugger

1. Introduction

The unique and important role of JavaScript in web programming is undeniable. Along with the wave of "Web 2.0", JavaScript has become the inevitable part of almost every modern web site. This language is used by the Web's 100 most popular sites[1] [16]. It is very likely that JavaScript keeps this crucial role for the next few years or even the next decade. Along with the growth of demands for more comprehensive user interfaces, the size and the complexity of web applications is increasing. Moreover, JavaScript is also becoming a general purpose computing platform[17] for office applications [14, 15], browsers [9, 10] and even program development environments [13]. There are also proposals for employing JavaScript in server-side applications [5, 18].

To cope with these large and sophisticated systems, JavaScript developers turn to development tools. One prime example is a runtime debugger: the developer can halt a running program and examine the program state and execution call stack.

All of these tools need to express program artifacts in a compact way the developer can understand. For example, the debugger must present the execution call stack so the developer can understand which functions are currently active. Obviously a particularly good compact representation would be a name given by the developer in the source code. However, the JavaScript language itself does not require names for many program artifacts and – as we shall see – nameless or anonymous artifacts are more common than named ones. Anonymous artifacts prevent tools from communicating effectively with developers.

Among program artifacts, functions are central to understanding a JavaScript program. In addition to their role in the execution stack, they are first-class objects that are used for different purposes by developers; they may be used as an object constructor, a closure scope (module) or even passed as an argument in a function call. These functions can be defined and created without a name or identifier.

In this paper, we analyze ten large, well-known projects. We show that within these projects less than 7% of the function bodies are named. We analyze the syntactic constructs surrounding these function bodies and rationalize how developers think about the bodies in relation to the structure of the code.

We propose an automated approach based on extracted data from the source code for naming JavaScript functions.

[1] http://www.alexa.com

Project	Ver.	Description	Total	Named
Closure	r683	Google Web Library	9195	208(2%)
DoJo	1.5	JavaScript Toolkit	18676	2810(15%)
ExtJS	3.3.1	JavaScript Framework	37717	1184(3%)
Firebug	1.7	Web Development Tool	3424	406(11%)
jQuery	1.4.4	JavaScript Library	422	23(5%)
MochiKit	1.4.2	JavaScript Library	1866	37(1%)
MooTools	1.3	JavaScript Framework	625	7(1%)
Prototype	1.7	JavaScript Framework	645	203(31%)
Scriptaculous	1.9	JavaScript Library	1092	208(19%)
YUI	3.3	Yahoo UI Library	22346	922(4%)
All		All Projects	96008	6008(6.3%)

Table 1. The total number of functions and the number of named functions (and percent named) in ten large JavaScript projects. See appendix 1 for the project citations.

The candidate function names can be used in debuggers for more descriptive object summaries and call-stack views, or in integration with proposed JavaScript typing systems for providing modern editing features in development environments.

2. The Anonymous Function Problem

A JavaScript function can be defined with the `function` operator or the `Function` constructor (i.e., `new Function( args, source)`) [7]. The `function` operator can appear in a function declaration, function expression, or function statement (the latter form is available in Firefox but not part of a standard). Of these forms, only the function declaration requires a function name and the `Function` constructor has no mechanism to name the function. In other words, JavaScript developers can define functions with or without function names.

If you are unfamiliar with JavaScript or other functional programming languages, you might imagine that developers would naturally select the form with names, simply as an organizational tool. However this is not the case. Functions in JavaScript are first-class objects. They can be assigned to any variable or object property, or passed as an argument to a function. Consequently JavaScript programmers can use these other constructs to organize their thinking about the program, without the use of function names.

So what do JavaScript developers do in practice? To get empirical evidence we analyzed the source code of ten well-known JavaScript projects[2]. For every project, the total number of functions and the number of functions with an identifier shown in Table 1. The average ratio of named functions to all functions is less then 7 percent and, excepting one project, *Prototype*, the ratio does not exceed 13 percent. Among all functions only a very limited number of them (116 functions) are defined by the `Function` constructor. Our analysis does not include the functions defined dynamically by `eval` function or `new Function()`; these cases would only make the ratio even smaller. Therefore, we conclude that a large proportion of JavaScript functions are anonymous.

[2] We did not perform any preprocess to exclude third-party or repeated files in the provided source code bundles.

```
9  var main = function() {              // main
10   var foo = new Foo(
11     function(){                      // main/foo<
12       this.welcome = "Hi!";
13     });
14   var bar = new Bar("GoodBye.");
15   alert(foo.welcome);
16   alert(bar.message);
17 };
18 var Foo = function(){                // Foo<
19   var instances;
20   return function(initializer){      // Foo
21     instances++;
22     initializer.apply(this);
23   }
24 }();
25 var Baz = Bar = function(msg){       // Bar
26   this.message = msg;
27 }
```

Figure 1. An excerpt of a JavaScript code illustrating anonymous functions. The comments give the results from Sec. 5 and these are discussed in Sec. 5.5

To understand the consequences of anonymous functions on development tools we will focus on one example, the impact on debuggers. Two main issues appear in debuggers due to the lack of function name. First, the object constructor name, which can facilitate understanding the object value, is not available in the object summary. Second, the call-stack view is usually full of *anonymous* functions and therefore much less informative. We discuss these issues in the next two subsections.

2.1 Missed Constructor Name in Object Summary

JavaScript does not support classes, but objects can be created by constructors (`new` followed by a function call). A constructor is a regular JavaScript function. Once the `new` keyword is evaluated, an empty object, with the constructor prototype as its prototype, is created, then the new object is bound to `this` and the constructor is called. The role of constructor is to initialize the empty object. Unlike class-based object-oriented languages, the structure of the object may change during the object's lifetime [17]. Nevertheless, the constructor can still be useful in classifying the object most of the time. Debuggers employ this fact and display the constructor name in the object summary to facilitate the developer's understanding.

Figure 1 shows an excerpt of a JavaScript program we use to illustrate this issue. We set a breakpoint on line 15 and examine the runtime elements at this breakpoint by two JavaScript debuggers, Google Chrome and Firebug (Figure 2). Two objects assigned to variables `foo` and `bar` are constructed by two different constructors: `Foo` and `Bar`.

The Google Chrome debugger shows the general class of `Object` for `foo` and the `Baz.Bar` class for `bar` in their summaries. The first class is very general and the second

(a) Google Chrome Debugger (b) Firebug

Figure 2. The screenshot of variables view of Google Chrome and Firebug JavaScript debuggers paused on a breakpoint at line 15 of the program shown in Figure 1. The content of these views is discussed in Sec. 2.1

one is misleading. The developer has to expand the object nodes to recognize their similarities and differences.

Firebug classifies both objects in the general `Object` class, but includes some of the object properties in the summary. These additional properties may give a hint to the developer about the object structures. `Foo` and `Bar` definitions at lines 20 and 25 explain the debuggers' behavior: the function statements has no explicit name (identifier), therefore debuggers considered them as anonymous functions or they infer a misleading name.

2.2 Anonymous Function Names in Call-stack View

The second problem is the call-stack. To illustrate this, we pause the program (Figure 1) at line 12 by a breakpoint. Figure 3 shows how the program call-stack is displayed in Google Chrome debugger and Firebug. The differences in the number of frames and line numbers between two call-stacks are due to dissimilar event handling implementations in the underlying platforms. Among the three top functions in the call stack, Google Chrome shows only the name of the third one down, function (`main`), correctly. For the second frame from the top (marked as line 22), it shows `anonymous`, and for the top frame (marked at line 12), it gives a wrong name, `foo`. Firebug performs better by guessing two function names correctly, but it still fails in one case. It shows `main` as the name of the function on the top frame (marked at line 12) but this is the name of another function (the enclosing function at line 9). In these cases, the information provided by the debugger is useless and the developer has to locate the function source to understand or recall the function behavior.

3. Automated Function Naming

The anonymous functions problem is discussed in several articles and forums on the Web [6, 20]. Different solutions have been proposed and discussed by practitioners. A basic solution is a mechanism for naming functions by developers without affecting the variables in scopes. For example, a new property (e.g., *displayName*) in the function object can be used for storing the function name, or the function name can be defined by an annotation. Although these solutions may help, they require extra work from developers, the displayName value can become out of sync with the meaning of the code over time, the annotation may be incorrectly recognized in programs that use the same property name for another purpose, and maintenance of the debugger becomes more difficult once the call-stack names can be overridden by user code.

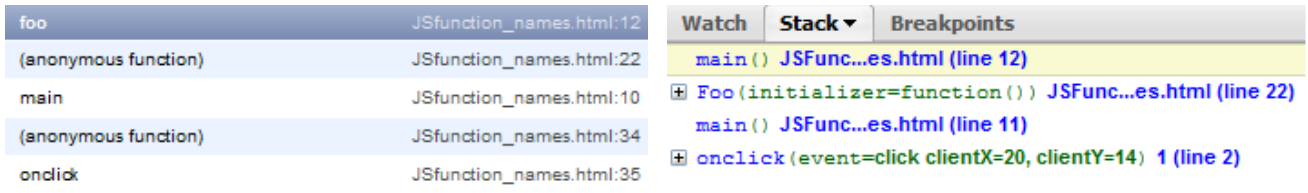
We instead propose an automated approach for naming anonymous functions by analyzing the source code. Before getting into explaining the algorithm details we discuss the rationale behind some of decisions we made in this approach.

3.1 What Should Be Named?

A statement defining a function creates new `Function` object. The definition may be evaluated multiple times, and depending on the times a function definition is evaluated, zero to many `Function` objects can be created from the same definition. Two function objects which are created from the same function body may have different object properties added at runtime. They may also have different enclosing scopes and therefore different behaviors. Thus our first question: do we try to name the `Function` objects or the source that defines them?

For the common cases, the different `Function` objects are bound to one or more properties of objects. The names of these properties inform the developer about the role of the function in the actions of the object. To determine the actions of the functions in turn, the developer must read the function source (or perhaps its documentation). Our function names serve to recall or summarize that source or documentation for the developer. Therefore we seek to name the source, the content between the curly braces known as the *FunctionBody* in the standard[7].

After reflection the reader may be puzzled by the preceding claim. On the one hand we claim that the function object instance may be bound to properties in multiple objects and those property names are not helpful for naming. On the other hand we will shortly introduce an algorithm that uses a property name (in part) to name a *FunctionBody*. Ultimately we are relying on a subtle characteristic of JavaScript programming: the first binding of a *Function* to an object property differs from all other bindings because it is located in text near the *FunctionBody* and thus developers associate this first binding with the meaning of the *FunctionBody*.



(a) Google Chrome Debugger (b) Firebug

Figure 3. The screenshot of call-stack view of Google Chrome and Firebug JavaScript debuggers paused on a breakpoint at line 15 of the program shown in Figure 1. The contents of these views are discussed in Sec. 2.2

3.2 What Makes a Good Function Name?

A function name is basically used to assist the developer to recall or understand the function behavior. For a developer who is already familiar with the function, it works more like an identifier. However, this identifier should be easily recognized by the developer. For example, a naive proposal for the function name is a combination of the function file name and its first line number. Although it may work as an identifier, it does not assist the developer to recall or understand the function behavior.

On the other hand, for a developer who does not know the function, a function name should explain the function behavior, or an abstraction of the function behavior, or why/where the function is used/defined (e.g., to create the object *foo*). A function name must not be so long that it can not be displayed or read by the developer. For example, the entire function body source code explains the function behavior well, however it is not an appropriate function name.

3.3 Context, Package and Function Names

JavaScript does not support a standard packaging mechanism to be used for modular programming. Scripts are loaded from different files and executed within the same or different global objects. Developers usually use objects at the top level to encapsulate objects, properties and functions from a framework or library. The same mechanism is reused for defining subpackages. As the project size and the number of functions increases, the short function name will not be enough for recognizing the function. The developer also wants the class or the module that contains the function. In addition to the package name, knowing the context (the enclosing function body) that contains the function can help in better understanding the function behavior.

4. Building Up Intuition by Example

We know that we face an ill-defined task: we are after all attempting to create short useful names for nameless functions. We have to create salient information from source code: we anticipate that removing characters will be our biggest challenge. To create an algorithm we decided to study the spectrum of examples from our 10 large collections of functions. We want to see what kinds of cases are

	Description				Code
1	The function object	is assigned to a(n)	property of a new object in an object literal.		{ ..., foo: function(){...}, ...}
2			new array index in an array literal.		[..., function(){...}, ...]
3			object property through	direct access by a property identifier.	bar*.foo = function(){...}
4				hashmap access by a string.	bar*["foo"] = function(){...}
5				hashmap access by a variable name.	bar*[foo] = function(){...}
6				hashmap access by a JavaScript expression.	bar*[foo*] = function(){...}
7			array index.		foo*[0] = function(){...}
8			variable.		foo = function(){...}
9		is directly called.			function(){...}()
10		property is accessed.			function(){...}.foo
11		is returned from a function call.			{... return function(){...}}
12		is passed as an argument to a function.			foo*(..., function(){}, ...)

Table 2. Different cases of anonymous function object creation and usage in JavaScript. Identifiers with a star in the table can be expressions as well as simple identifiers; we explain how we reduce expressions to pseudo-identifier in 5.3.

important and what aspects of these cases help us identify functions.

For this purpose we created 12 categories of function body expressions shown in Table 2 and we categorized all of the nameless functions from the 10 JavaScript projects into one of these 12 cases, giving the numerical results in Table 3. Then we examine each of these cases to think about how we want the functions named. We will skip over the nesting of function scopes in the analysis to avoid taking on too much at one time; we return to this aspect at the end of this section.

4.1 Case 1: Object Property Initializer

This case usually appears when developers tries to group a set of functions in a new object. A common case is grouping a set of functions in the `prototype` property of the constructor. This structure resembles the class structure in traditional object-oriented languages. When a new object is created by the constructor, the new object also inherits all functions defined in the constructor's prototype. This structure is also used when the owner object is a shared object with a set of utility functions.

The majority of nameless functions (more than 65%) in almost all studied projects (except Closure), are defined in object literals. This case seems particularly simple: the property name makes a good name for the function body. But what logic are we implicitly applying here? Our reasoning: the developer-invented property name has high information content, it is textually close to the function body, and the function object created by the function body initializes to the property. These observations guide us in more complex cases.

4.2 Case 2: Entry in an Array Literal

Contrary to the previous case the appearance of this case is very limited. It usually appears in initializations or when an array of functions are passed as an argument. Among all projects only three have instances of nameless functions defined in this way. For example array argument to `Event._attach` in this example from YUI [3]:

```
_attach: function (el, notifier, delegate) {
  if (Y.DOM.isWindow(el)) {
    return Event._attach([type, function (e) {
      notifier.fire(e);
    }, el]);
  }
}
```

The developer will probably think of the array entry as one item in a collection passed to `Event._attach`. In general we shall want to name these kinds of functions by the destiny of the containing array.

4.3 Case 3: Property Assignment With Property Identifier

This case is the second most common case in the studied projects. Here, we can see why the Closure project is different from other projects in the first case. About 94% of nameless functions in this project are defined in this way. It seems that the Closure developers follow an internal standard for function objects creation and usage.

Following the model from Case 1, we think a good name would combine the object name with the property identifier. The complication in this case comes from the object name: in general the object reference can be a computed expression[4]. Here is a simple example from the Closure project:

```
this.eventPool_.createObject = function() {
  return new goog.debug.Trace_.Event_();
};
```

In general, the expression can be long and complex: to create a useful name we need to focus on developer-invented identifiers in the expression and work to keep the total number of characters small. For example, `this.` add no information to the name since we cannot know the value of `this` while parsing.

4.4 Case 4: Property Assignment With Property Name String

In JavaScript, objects are like hashmaps and their properties can also be accessed by a string specified in the brackets after the object. Semantically this is the same as the previous case and we see few instances of this form of function object assignment. The string inside the brackets can be considered a property identifier for naming.

4.5 Case 5: Property Assignment With Property Name Variable

Object member names can be variable references that get converted to strings at runtime: this is syntactically similar to Case 4, but we cannot (usually) statically compute the string to use as a property identifier. The usage of this case is also limited. This form usually appears when the same function body is assigned to different properties in a loop. For example see the inner function in Fig. 4. The variable name in the cases we examined was generic name like `o` or `item`. Unlike the previous two cases, we do not have a specific property name. Nevertheless identifying the function body using the assignment target with the variable name as the property name follows the reasoning used for the simple cases.

4.6 Case 6: Property Assignment With Property Name Expressions

Object member names can be expressions that get converted to strings at runtime: this is more general than case 5. A common case of expression in this case is a conditional expression, e.g., `condition?"prop1":"prop2"`, where the property that we assign the function to depends upon runtime values. Another kind of example of computed names comes from the Prototype project (reformatted to fit in the page):

[3] The example is nested in more function definitions we do not show here.

[4] This comment applies to all of the cases in table 2 marked with an asterisk on the expression identifier

	1	2	3	4	5	6	7	8	9	10	11	12
Closure	23(0.3%)	0	8466(94%)	3	4	0	0	67(0.7%)	16(0.2%)	0	43(0.5%)	365(4%)
DOJO	9601(61%)	7	1765(11%)	21	24	2	1	1151(7%)	476(3%)	2	175(1%)	2641(17%)
ExtJS	30221(83%)	0	1476(4%)	3	40	9	0	859(2%)	788(2%)	180	517(1%)	2439(7%)
Firebug	2296(76%)	0	539(18%)	1	2	0	0	17(0.4%)	7(0.2%)	2	6(0.2%)	148(5%)
jQuery	233(58%)	0	34(9%)	0	10	2	0	24(6%)	10(2%)	0	0	86(21%)
MochiKit	1080(59%)	10	385(21%)	0	4	0	0	110(6%)	18(1%)	0	41(2%)	181(10%)
MooTools	339(55%)	0	79(13%)	0	4	3	0	53(9%)	21(3%)	20	14(2%)	85(13%)
Prototype	265(60%)	0	28(6%)	0	0	1	0	25(6%)	44(10%)	1	8(2%)	70(16%)
Scriptaculous	564(64%)	0	75(8%)	0	2	2	0	27(3%)	45(5%)	21	9(1%)	139(15%)
YUI	14154(66%)	7	1721(8%)	0	90	0	0	1181(6%)	172(1%)	0	95(0.5%)	4004(19%)
All	58776(65%)	24	14568(16%)	28	180	19	1	3514(4%)	1597(2%)	226	908(1%)	10072(11%)

Table 3. The number of nameless functions in each category defined in table 2.

```
jQuery.each("ajaxStart ajaxStop ajaxComplete ajaxError"
  .split(" "), function( i, o ) {
    jQuery.fn[o] = function( f ) {
      return this.bind(o, f);
    };
});
```

Figure 4. An example of a function (the inner definition) assigned to a hashmap using a variable name (row 5 in Table 2) and an example of functions passed as arguments to a function (row 11 in Table 2). The function is from the jQuery library but simplified to fit on the page.

```
function define(D) {
  if (!element) element = getRootElement();
  property[D] = 'client' + D;
  viewport['get' + D] = function() {
    return element[property[D]]
  };
  return viewport['get' + D]();
}
```

The word `get` is concatenated with the `toString()` value of the argument `D` at runtime to create the property name. Unlike Case 5, the property name expression need not be a simple developer-invented identifier. In this example, `viewport[getD]` could be a good name, but in general we will need to process the expression to balance length with information.

Notice that from the programming language point of view, Cases 3 through 6 are all special cases of Case 6. After all we are just selecting an object property in all of these cases. But from a naming point of view these cases present different challenges and the more complex cases will make our necessary tradeoffs more costly.

4.7 Case 7: Assignment to an Element of an Array

We only observed one instance of this form in the studied projects. The numerical index should clearly be part of the name; the array name may be an expression that we have to analyze to create name.

4.8 Case 8: Assignment to a Variable

This case is widely used and we expect developers would expect the function body to get the name of the variable. As the functions are immutable objects, it is very likely that a function object which is assigned to a variable, is used with the same variable name in the function scope and its internal scopes. There are cases in which a variable name is used temporarily, as the function is passed to another function or assigned to an object property. However, in most cases the variable name works well as the function name.

4.9 Case 9: Anonymous Functions Immediately Called

Calling function objects just after their creations is a common pattern in JavaScript. For example see the assignment to `Y.ClassNameManager` in Fig. 5 (the function is called at the bottom of the example).

```
Y.ClassNameManager = function () {
  var sPrefix = CONFIG[CLASS_NAME_PREFIX],
  sDelimiter = CONFIG[CLASS_NAME_DELIMITER];
  return {
    getClassName: Y.cached(function () {
      var args = Y.Array(arguments);
      if (args[args.length-1] !== true) {
        args.unshift(sPrefix);
      } else {
        args.pop();
      }
      return args.join(sDelimiter);
    })
  };
}();
```

Figure 5. An example the YUI project of function bodies from cases 9 (the outer function) and 12 (the argument to Y.cached) from Table 2

If the called-function is assigned to a variable or object property, then we have a version of one of the other cases in Table 2. The difference here is that we can tell from static analysis that the assignment will use the return value of the function, not the function object itself. But for naming purposes the key information will be the assignment target.

If the function call has no result, it means that the function performs one task (e.g., initialization of some values in the outer scope for later use). In this case we cannot use the assignment target idea from Case 1, but the source proximity and the developer-invented names concepts point to using interior identifiers in a name. To avoid confusing the developer

by using the same name for the outer and interior functions, we will need some way to signify that the name we create in this way is for an immediately called function.

4.10 Case 10: Function Property is Accessed

In this unusual case a property of the function is accessed directly from the function body. This case usually happens when one of the predefined functions (i.e., `call, apply, bind`) or added functions to the `Function` prototype is called. Here is an example, from ExtJS, reformatted for display here:

```
setVisible : function(v, a, d, c, e){
  if(v){
    this.showAction();
  }
  if(a && v){
    var cb = function(){
      this.sync(true);
        if(c){
          c();
        }
    }.createDelegate(this);
  }
  // .....
}
```

The inner function body is not used directly, but the result of `createDelegate` is assigned to `cb`. The most valuable information here is the variable name `cb`, followed by the `createDelegate` function name. This example also illustrates that automatic naming could have an effect on developers coding style: giving a longer name for `cb` would give better names in development tools but currently developers have limited expectations that tools will show such information.

4.11 Case 11: Returned From a Function Call

In this case the function body appears in a return statement of another function body. Although this category only includes 2% of nameless functions, proper naming of functions in this class is important. Many constructors are built using this form and therefore the names of these functions appear in object summaries. Clearly the name of these returned functions is almost the same as the name of the functions that define them. For example, in Fig. 6 a developer might pick names like `registerWinOnIE` and `registerWinNotOnIE` for the functions returned by the Dojo function `registerWin`.

4.12 Case 12: Function Passed as an Argument

Numbers in table 3 show that creating and passing functions as arguments is very common in JavaScript. For example, see Fig. 4. The calling function and the other arguments look helpful, to the extent that they have identifiers invented by the developer. As in this example, we see that the calling function, `jQuery.each()`, can be generic so it provides less valuable information, but the arguments in that example are highly specific to the function body. Fig. 5 shows different example, where the function called (`Y.Cached()`)

```
registerWin: function(targetWindow, effectiveNode){
 ...
 if(doc){
  if(dojo.isIE){
   ....
   return function(){
    doc.detachEvent('onmousedown', mousedownListener);
    doc.detachEvent('onactivate', activateListener);
    doc.detachEvent('ondeactivate', deactivateListener);
    doc = null;
   };
  }else{
   ....
   return function(){
    doc.removeEventListener(
                    'mousedown', mousedownListener, true);
    doc.removeEventListener('focus', focusListener, true);
    doc.removeEventListener('blur', blurListener, true);
    doc = null;
   };
}}},
```

Figure 6. An example of a function body in a return statement, case 11 of Table 2 adapted from the Dojo project code

seems much less important for naming the function than the property that we initialize with the result of calling the function. This important case will stress any naming algorithm: we somehow have to summarize the function call – which itself may be an expression – and the other arguments – any or all of which may be expressions.

4.13 Results of Studying Examples

We reached two main conclusions from studying the way anonymous functions are used in the source of the 10 projects we examined. First, we want to try to find the name of the initializer or assignment target that will receive the function object created from a function body. JavaScript programmers are creating anonymous functions but they are loading them into object references and the expressions that result in those references have informative identifiers inside. Second, these expressions that we focus on may often be simple identifiers, but if they are not we will need to analyze the source code of these expressions to extract meaningful summaries. This summary has to balance information against length.

Overlaying our analysis above is hierarchy: any of the cases can be nested in function scopes. Obviously this hierarchy must be represented in our names. Algorithmically this is straight forward recursion. But from the name usability point of view, deep hierarchy means long names, exactly the problem we want to avoid. Fortunately developers are well trained in dealing with this kind of problem and we anticipate that hierarchical names can be shown to users in progressive depth depending on the particular needs in the user interface.

5. Static Function Object Consumption

Using our analysis we have created a preliminary automatic naming solution. The three parts of our solution match three

observations for our study. First we apply *(Static) Function Object Consumption*, which tracks the function object created from a function body to where the object is 'consumed', for example by assignment to or initialization of an object property, variable reference, or function argument. The "static" qualifier just indicates that we will only use a parser. Second, we reduce complex expressions to pseudo-identifiers focusing on developer-invented names. Third we apply our approach hierarchically to deal with nested function bodies.

We will describe the details in the next sections. In constructing names we realized one additional aspect: as we move from simple object property names to more complex examples, the path of the object consumption is an added bit of information helpful in naming. Thus we add some symbols to guide the developer to the function body in complex cases: in this way the extra information does not take a lot of space and it can be ignored by developers who have not yet learned about its significance. We describe these symbols in Sec. 5.4.

5.1 Consumption Summary Algorithm

We parse the JavaScript and search the resulting syntax tree for function body nodes. For each function body, we apply the algorithm outlined in Algorithm 1. The basic idea (the `while` loop) is to walk the syntax tree from the body up through parent nodes until we hit a node that is not a JavaScript expression. For each node we create an entry in a list and record in it information about the relationship between the node and its parent. We'll use this relationship information in Sec. 5.4. Then for each node we record developer-invented identifiers related to the destiny of the function object created from the body as outlined in Table 4. For the first and third rows of the table we record the information recursively; for the second row, assignment node, we record the information after the loop terminates. The algorithm ends when we reach a node which is an assignment or a statement which does not return any value (i.e., the node is not an expression). The algorithm result is an *object consumption summary*, a list of collected data at every visited parent node of the function body.

The algorithm uses three subroutines: `getNextNode`, `nameExpression`, and `argSummary`. The `getNextNode` routine normally returns `n.parent`, but we also use this point in the algorithm to handle an important special case, function bodies inside of *immediate functions* typically used for modularity or scoping in JavaScript. We discuss this case in Sec. 5.2. The remaining two subroutines construct a pseudo-identifier from the expression as described in 5.3; they return their argument if it is simply an identifier.

5.2 Consumption by Immediate Functions

JavaScript developers use function scope to dynamically create functions with shared but private state. In this pattern, an enclosing function contains a number of function and

Algorithm 1 Compute Object Consumption Summary for Function Body Nodes

Input: Function Body Node n in Abstract Syntax Tree
Output: Object Consumption Summary

```
List summary = new List()
while n.parent is an expression do
  dataItem = new DataItem()
  if n value is the same as n.parent value then
    dataItem.isTheSame = true
  else if n value is a property of n.parent value then
    dataItem.isPartOf = true
  else
    dataItem.isContributesTo = true
  end if
  if n.parent is a function call and n is an argument then
    dataItem.isFunctionCall = true
    dataItem.id = nameExpression(n.parent)
    dataItem.hint = argSummary(n.parent)
  else if n.parent is an object literal and n is assigned to
  foo then
    dataItem.isObjLiteral = true
    dataItem.id = foo
  end if
  summary.add(dataItem)
  n = getNextNode(n)
end while
if n.parent is an assignment then
  dataItem = new DataItem()
  dataItem.isAssignment = true
  dataItem.id = nameExpression(n.parent)
  summary.add(dataItem)
end if
return summary
```

		Description	Code
1	The parent node	is an object literal.	{ ..., foo: expr }
2		is a function call.	foo*(..., expr, ...)
3		is an assignment.	foo* = expr

Table 4. Nodes produce identifiers in the function object consumption summary. Identifiers with a star in the table can be expressions as well as simple identifiers; we explain how we reduce expressions to pseudo-identifier in 5.3.

object definitions and it is called immediately after it is defined. We call these functions *immediate functions*. For an example, see line 10 of the Figure 1 which is enclosed in the function on line 9. If the developer returns a function from this outer, immediate function, we want to follow the returned object to where it is consumed.

To keep our core algorithm simple we handle these returns from immediate functions as a special case. At the end of each loop in Algorithm 1 at the point marked `getNextNode()` we check to see if the parent is a `return` node. If not we return the parent node and the loop continues. If we have a `return` node, we look to the parent of the `return` node to see if it is a call (either directly with `()` or via `apply()` or `call()`. If so, we know we are returning a function object from an immediate function. We return the parent of the immediate function, effectively skipping the intermediate nodes so that the name will reflect the consumption of the return value into the destination of the immediate function.

5.3 Expression Reductions

In simple cases the syntax tree node will have an identifier we can use as part of our name. In more complex cases an expression will be written in place of an identifier. We reduce these expressions to pseudo-identifiers (i.e. not necessarily a valid JavaScript identifier) that resembles the expression. This work is done during the Object Consumption Summary algorithm in the functions described here:

Identifiers for Function Arguments Search for all literal string nodes in other arguments and concatenated them by "-", dropping characters beyond 10. (This work is on in Algorithm 1 in `argSummary`).

Identifiers for Assignments and Function calls This function gets called for nodes resulting from the last 2 rows of Table 4. The expressions here will evaluate to a writable or readable address, so we want to extract the most specific developer-invented identifiers from the expression. Thus we apply the rules from Table 5, which starts from the right hand side of the expression; we skip any JavaScript keywords like `this` or `prototype`. We also skip any pattern which does not match (e.g., a function call). (This work is on in Algorithm 1 in `nameExpressions`.)

Obviously these rules are heuristic and can be improved through experience and interaction with developers. We are attempting to balance information content with length. Large complex expressions will give pseudo-identifiers which are complex, but with some identifiable parts adequate for developer recognition and search.

5.4 Conversion to an Name

At this stage of the algorithm we have a list of identifiers with attributes which we want to concatenate to create a name.

Description	Pattern	Name
Primitive	value	value.toString()
Variable	id	id
GetProp	e.id	Name(e).id
GetElem	e1[e2]	Name(e1)+[+Name(e2)+]
Operation	e1 op e2	Name(e1)+op+Name(e2)
Condition	cond?e1:e2	Name(e1)+:+Name(e2)

Table 5. JavaScript Expression Reduction to a Name. Expressions which match an entry in the pattern column are converted as shown in the Name column. Here `e` indicates an expression, `id` indicates an identifier, + means string concatenation and `Name()` means we apply the pattern matching recursively.

First we drop function-call identifiers if we have anything else to use for a name. The function-call identifiers typically tell us about a transformation of the function body before it is assigned to an object property or variable. The transformation may be used many places in the code, while the assignment target is typically an identifier defined by the developer for a specific section of source code. See, for example, function on line 14 and the function call on line 13 in the Figure 1. Specifically we drop identifiers from row 2 of table 4 in any case where we have identifiers from rows 1 or 3. The outer function in Fig. 4 illustrates the opposite case, where we do not drop function-call identifier.

Second we concatenate the identifiers with a symbol between each parent and child showing the relationship. As shown in Algorithm 1, if the parent expression is an array or object literal then the identifier will be marked as *isPartOf* and we insert a dot character. If the identifier was marked as *isContributesTo* we insert a left angle bracket. An identifier marked *isSameAs* is skipped over because we already have identifier information for it. Because some entries on the object consumption list are empty strings we may have duplicate symbols. Any duplicates are replaced by a single symbol. The result contains identifiers (i.e., variable and property names) and strings available in the source code plus some explanatory tokens. It compactly explains the function object creation, consumption and assignment.

The particular symbols we use may be refined with more experience in how developers respond to them. Our intuition is that these extra symbols need to be visually compact because their purpose is to adjust the developers expectation for the identifier. For example, the function on line 9 of Fig. 1 is named `main<` but the developer may only key on the word `main` to recall the function body.

The name built by the above process does not contain any information about enclosing scopes. We call it *local name*. We get the function *full name* (that is a local name qualified by its position in the scope hierarchy) by adding the enclosing function full-name with a slash before the local-name, recursing through enclosing scopes. This full-name is the function body name.

5.5 Examples

To further explain our approach we now apply it to the JavaScript code presented in Figure 1. We can recognize five function bodies in the code, none of them has a name.

The first function on line 9 is assigned to the variable `main`. The parent node for the function body in the syntax tree will be an assignment, so in Algorithm 1 we skip the `while` loop and compute the `nameExpression` as `main`. Ultimately this becomes the name of the function.

The second function on line 11 is nested in the first one. It is passed as an argument to a constructor call on line 10 and the resulting object is assigned to `foo`. In Algorithm 1 we create two entries in the consumption summary, one for the function call with identifier `Foo` and one for the assignment with identifier `foo`. The first one gets marked with `isContributesTo`. Following the logic in Sec. 5.4, we drop the function call identifier but mark the name with < for contributes-to. The resulting local name is `foo<` and the full name includes the enclosing function name: `main/foo<`.

The third function on line 18 is called immediately after definition, on line 24. This means the function body in the syntax tree will have a parent from the immediate call and then the assignment parent node; the call does not give us an identifier but it does give a consumption summary entry with `isContributesTo`. The name becomes `Foo<`.

The fourth function on line 20 is returned by `Foo<`. After the first pass through the `while` loop in Algorithm 1 we enter `getNextNode` and trigger the code described in Sec. 5.2. This will cause us to walk up the syntax tree to find the assignment target for the outer function. We end up with name `Foo`. Because do not process the intermediate nodes in the `while` loop we do not mark the name with contributes-to and we do not record that the function is nested. Of course the function body is nested, but it is bound to an un-nested variable. To keep the name compact we choose not to encode this complex information. Rather we stick to the simple picture that the function 'is' `Foo`. If the developer wants to know more they can look up `Foo` to see the construction and nesting.

The fifth function on line 25 is assigned to two variables. This example illustrates that we stop processing in Algorithm 1 as soon as we hit the first assignment, giving the name `Bar`. This aligns with our observation that the visually closest identifier is the best choice.

Finally, Table 6 gives the names of functions from examples discussed in 4.

Example Code	Static Function-Object Consumption Full Name
Sec. 4.2 outer	YUI.add(event-focus)/_attach
Sec. 4.2 inner	YUI.add(event-focus)/_attach/Event._attach()
Sec. 4.3	eventPool._createObject
Fig. 4 outer	jQuery.each(ajaxStart)
Fig. 4 inner	jQuery.each(ajaxStart)/jQuery.fn[o]
Sec. 4.6	define/viewport[get+D]
Fig. 5 outer	YUI.add(classnamem)/Y.ClassNameManager<
Fig. 5 inner	YUI.add(classnamem)/Y.ClassNameManager.getClassName<
Sec. 4.10 inner	setVisible/cb<
Fig. 6 inner	registerWin/

Table 6. Results from applying the approach in Sec. 5 to the examples in the paper at the point given in the first column. Full names are listed, even in cases where our example code omitted the enclosing function scope.

6. Evaluation

In Table 7 we compared our results to Firebug's naming output for each of the 10 projects used previously and listed in Appendix 1. Firebug does not use a parser for naming functions. It instead employs a number of regular expressions and apply them on a few lines around the function definition to obtain a name (a function called `guessFunctionName()`). Although this approach is not reliable and may provide wrong names for some functions, it infers acceptable names in many simple cases. About 10 percent (the second column) of anonymous functions still remained nameless in Firebug (in addition, some named functions may be quite incorrect because of the simple algorithm). The third column shows that more than 98 percent of anonymous functions have a FOC-defined local name. By adding the enclosing function names, this number increases to more than 99 percent for FOC-Full(the fourth column). Based on our observations, most functions remaining anonymous in FOC-Full are top level *immediate* functions which are usually used for creating a local scope for a set of variables.

The "Duplicates By" columns show the number of functions which get a name which is also assigned to another function(s) for Firebug, FOC-Local and FOC-Full. The number shows that Firebug assigns duplicate names to about 47 percent of anonymous functions. The first reason behind this huge number is that Firebug relies on regular expressions instead of abstract syntax tree for locating names. The second reason is that Firebug does not analyze the function object flow but tries to construct the name from the identifiers close to function definition. The second column shows that FOC-Local gives much less duplicates (13%) comparing to Firebug. This means that local names are sufficient for recognizing functions within a file, which is helpful because they have shorter length than full names. The third column says that less than 9 percent of anonymous functions get duplicated names by full names. Based on our observations, many of duplicates in projects such as DOJO, ExtJS and YUI come from test files. In a test file the same use case is repeated with different data and therefore the same function names appear several times. In projects like JQuery, Prototype, MooTools the conditional statements are the main source of duplicates. These libraries usually check against different environment properties (e.g., browser) to load the appropriate function.

The last two columns show the average and longest function name lengths (in characters) for both short and full names. The average length is at most 37 characters. The

last column shows that only 3 percent of functions get local names longer than 50 characters or full names longer than 80 characters. Again, based on our observations a main source of long names are deep nested functions which mostly appear in test files.

7. Discussion

Overall the Function-Object Consumption approach dramatically reduces the number of functions that a development tool cannot name for a developer. By looking at the examples and comparing the algorithms, the names selected by FOC will be the same as the one's selected by Firebug's `guessFunctionName()` when that function gives reasonable results. In other cases FOC will give a more useful name including many cases where the current Firebug approach fails.

There is room for improvements. The heuristic elements of the naming solution need to be tuned based on more experience. Given that our goals trade precision for rapid recognition, our names are not unique. We also leave some cases anonymous which need further investigation.

A complete naming solution will need to consider some addition issues. The user interface that shows names typically has limited space and even our efforts to create short names may be adequate. Often the hierarchy can help: we can show the trailing entry in a slash delimited list and the trailing entry in a dot delimited list with mouse-over expansion to bring up an overlay line with more of the full name displayed. Some JavaScript libraries have a regular pattern or specific re-naming registration system (see for example the ExtJS ClassManager[8]). A naming solution should recognize these patterns or systems. Similarly JavaScript libraries might support the `.displayName` to provide custom names where the developer overhead to introduce and maintain the extra field is justified by the wide spread use of the library.

8. Related Work

To best of our knowledge, this paper is the first study in naming anonymous functions in JavaScript. However, there have been a few studies on the JavaScript programs behavior. In a recent work, Richards et al. conducted a study on dynamic behavior of JavaScript programs [17]. Although their study contains some aspects of JavaScript function objects creation and usage, such as the number of different call sites and arities in function calls, it does not provide any data about anonymous functions.

We previously mentioned four JavaScript naming approaches: Zaytsev advocates expanded use of names in function expressions[20], support for `displayName` in debuggers has been requested[6], the Firebug Regular expressions, and the Google Chrome debugger's function name 'inference'[4]. The last one appears to be a parser based approach with goals similar to the ad-hoc Firebug regular expressions: look for identifiers preceding anonymous function bodies.

Høst and Østvold [12] attempted to find and fix inappropriate function names in Java programs. Their approach employs rules extracted from a large corpus of Java projects to recognize buggy names. To fix a name, a list of candidate names are constructed and ranked, the the name with highest rank is proposed for the replacement. This approach assumes that the function is already named and constructs the names based on the function internal structure. In contrast, our approach construct the function name by analyzing the function context.

Caprile and Tonella [3], analyze the structure of function identifiers in C programs. The identifiers are decomposed into fragments that are then classified into seven lexical categories. The structure of the function identifiers are further described by a hand-crafted grammar.

The result of our work can be improved and impact other aspects of JavaScript programming if a stronger typing system is employed. A few static typing systems have been proposed for JavaScript [1, 2, 11, 19]. These approaches discover the type of values and object structures for a variable by statically analyzing the source code and possible program control flows lead to the variable assignment. The discovered facts about variable types are not only useful for catching errors but to assist developers in code comprehension. Nevertheless, none of the mentioned approaches provided effective means for sharing this information with the developer. Our approach can help in this regard by assigning names to nameless elements.

9. Conclusion

Our contributions in this paper include an empirical study of the extent of anonymous functions in JavaScript libraries, categorization of those functions to understand the potential for automatic naming, an practical algorithm based on the empirical study, and its evaluation. We believe our result can be applied directly in existing JavaScript development tools to give immediate benefit to developers and provides a basis for future improvements.

A. JavaScript Projects Analyzed for Function Names

Closure Closure Library, r683, Google Code, http://code.google.com/p/closure-library/

Dojo Dojo JavaScript Toolkit, version 1.5, http://dojotoolkit.org/

ExtJS JavaScript Framework, version 3.3.1 http://www.sencha.com/products/extjs/

Firebug Web Page Debugger, version 1.7 http://code.google.com/p/fbug/.

jQuery JavaScript Library, version 1.4.4 http://jquery.com/

Project	Anonymous Functions By				Duplicates by			Local/Full Length		
	Developer	Firebug	FOC-Local	FOC-Full	Firebug	FOC-Local	FOC-Full	Average	Longest	>50/>80
Closure	8987	312(3%)	51	10	662(7%)	211(2%)	103(1%)	33/35	76/101	471/30
DoJo	15866	3362(21%)	528	390	3325(21%)	2993(19%)	2287(14%)	24/36	81/162	680/765
ExtJS	36533	1543(4%)	631	231	23777(65%)	3737(10%	2342(7%)	22/25	65/91	177/2
Firebug	3018	167(5%)	14	10	481(16%)	85(3%)	32(1%)	27/36	70/140	76/18
jQuery	399	95(24%)	9	8	162(40%)	97(24%)	56(14%)	14/18	50/65	0/0
MochiKit	1829	202(11%)	59	16	491(27%)	314(17%)	191(10%)	17/22	49/80	0/0
MooTools	618	121(19%)	30	18	272(44%)	158(26%)	138(22%)	12/14	36/73	0/0
Prototype	442	125(28%)	24	19	154(35%)	70(16%)	70(16%)	21/25	55/88	5/1
Scriptaculous	884	195(22%)	25	19	306(36%)	131(19%)	86(9%)	21/27	55/132	7/5
YUI	21424	3317(15%)	212	15	10073(47%)	3558(17%)	2205(10%)	16/37	114/145	223/292
All	90000	9439(10%)	1583(2%)	736(1%)	39703(44%)	11354(13%)	7510(8%)	N/A	N/A	1639/1113 (3%)

Table 7. Results of Function Object Consumption. The rows are the projects listed in Appendix 1. The first column contains the number of anonymous functions in each project. The second column shows the number of functions Firebug could not name. The third column contains the number of functions nameless after applying the static Function Object Consumption (FOC) algorithm; and the fourth column is the number of functions the FOC leaves without a name even from enclosing scopes. The next three columns give the number of times a Firebug, FOC-Local, and FOC-Full function name appears twice in a file, respectively. The last three columns contain the length information. Every cell has two entries divided by a slash, the first is the local name and the second is the full name. The eighth and ninth columns give the average and longest name character counts respectively. The last columns shows the number of functions with local names greater then 50 characters in length separated by a slash from the number of functions wit full names with length greater than 80 characters.

MochiKit version 1.4.2
http://mochi.github.com/mochikit/

MooTools JavaScript Framework, version 1.3
http://mootools.net/

Prototype JavaScript Framework, version 1.7
http://www.prototypejs.org/

Scriptaculous JavaScript Library, version 1.9
http://script.aculo.us/

YUI Library, Yahoo, Inc., version 3.3
http://developer.yahoo.com/yui/

References

[1] C. Anderson, P. Giannini, and S. Drossopoulou. Towards Type Inference for JavaScript. In *Proceedings of the 19th European conference on Object-Oriented Programming(ECOOP)*, July, 2005.

[2] C. Anderson and P. Giannini. Type checking for javascript. *Electr. Notes Theor. Comput. Sci.*, 138(2), 2005.

[3] B. Caprile and P. Tonella. Nomen est omen: Analyzing the language of function identifiers. In *Proceedings of the 6th working conference on Reverse Engineering(WCRE)*, October, 1999.

[4] V8 Google FuncNameInferrer http://v8.googlecode.com/svn/trunk/src/func-name-inferrer.h

[5] Common JS. http://www.commonjs.org/

[6] Add prettyName/displayName support to Profiler output and Stacks. *Firebug Bug Repository*, http://code.google.com/p/fbug/issues/detail?id=1811

[7] ECMA International. *ECMA-262: ECMAScript Language Specification*, ECMA (European Association for Standardizing Information and Communication Systems), Geneva, Switzerland, third edition, December 1999.

[8] Jacky Nguyen *Ext.ClassManager* http://dev.sencha.com/deploy/ext-4.0-beta2/docs/source/ClassManager.html

[9] Firefox Add-ons. https://addons.mozilla.org/en-US/developers/docs/getting-started.

[10] Google Chrome Extensions. http://code.google.com/chrome/extensions.

[11] P. Heidegger and P. Thiemann. Recency types for dynamically-typed, object-based languages. In *Proceedings of Foundations of Object Oriented Languages (FOOL)*, 2009.

[12] E. W. Høst and B. M. Østvold. Debugging Method Names. In *Proceedings of the 23rd European conference on Object-Oriented Programming(ECOOP)*, July, 2009.

[13] D. Ingalls, K. Palacz, S. Uhler and A. Taivalsaari. The lively kernel a self-supporting system on a web page. In *Self-Sustaining Systems*, 2008.

[14] JScript development in Microsoft Office 11. http://msdn.microsoft.com/en-us/library/aa202668(office.11).aspx

[15] JavaScript development in OpenOffice. http://framework.openoffice.org/scripting/release-0.2/javascript-devguide.html

[16] G. Richards, C. Hammer, B. Burg and J. Vitek. The Eval that Men Do. In *Proceedings of the 25th European conference on Object-Oriented Programming(ECOOP)*, July, 2011.

[17] G. Richards, S. Lebresne, B. Burg and J. Vitek. An analysis of the dynamic behavior of JavaScript programs. In *Proceedings of the 2010 ACM SIGPLAN conference on Programming language design and implementation(PLDI)*, June, 2010.

[18] Server-Side JavaScript Reference v1.2. http://research.nihonsoft.org/javascript/ServerReferenceJS12.

[19] P. Thiemann. Towards a type system for analyzing JavaScript programs. In *Proceedings of European Symposium on Programming (ESOP)*, 2005.

[20] J. Zaytsev. Named function expressions demystified. http://kangax.github.com/nfe, June, 2010.