# 2013W1-lecture1

September 12, 2013

## Contents

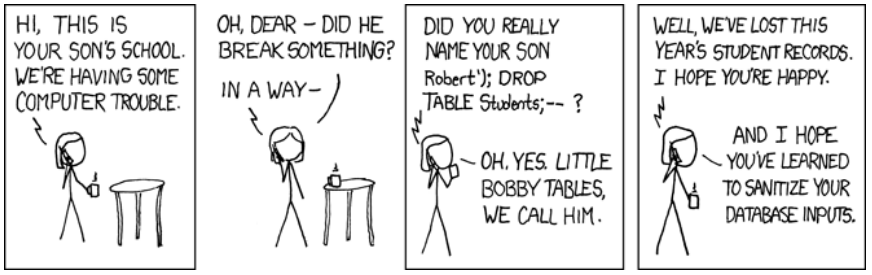
# 1 CPSC 311 Introduction

This is "Definition of Programming Languages". I'm Steve Wolfman. Call me Steve. Find me and say hello!

Meanwhile, what is the most common security vulnerability in programs as of:

- 2011
- 1995

Not this, but it's relevant:




(Answers: cross-site scripting and buffer overflow, respectively.)

## 1.1 Programming Languages are Magic Languages; Interpreters are Magic Machines; What's 311?

Programming is the closest thing to spell-casting we sad real-worlders get. We write words in arcane tongues; unseen processes cast them for us; and they change the world.

Programming languages are the arcane tongues. The unseen processes are compilers and interpreters: programs that take otherwise incomprehensible documents[1] and evaluate them.

Our job this term is to study the languages, *not* compilers/interpreters. CPSC 411 will study compilers.

But, what does it mean to study the languages? After all, you've already learned at least two languages.

## 1.2 Aside on Arrays: A Fundamental Programming Construct?

Let's consider the humble array. You've seen it in Java. It's less important in Racket but still there (vector).

[1] With the exception of literate programming.

### 1.2.1 Arrays: All the Same?

In *many* languages, arrays look about the same. Here's some C code with arrays:

```
int array[5];
int i;
for (i = 0; i < 5; i++) {
  array[i] = i*i + 1;
}
printf("array length is %d; array[3] = %d\n",
       (sizeof(array)/sizeof(array[0])), array[3]);
```

Here's similar JavaScript code:

```
var array = new Array(5);
for (var i = 0; i < 5; i++) {
  array[i] = i*i + 1;
}
console.log("array length is " + array.length +
            "; array[3] = " + array[3])
```

And here's similar C++ code (using the new C++11 standard):

```
std::array<int, 5> array;
for (int i = 0; i < 5; i++) {
  array[i] = i*i + 1;
}
std::cout << "array length is " << array.size() << "; array[3] = " << array[3];
```

Despite differences in syntax, all three use the *same* syntax for the inside of our loop, which isn't a coincidence. All three also allow you to find the array's length, although C's method is onerous!

However, the three constructs we're manipulating in these programs are *wildly* different things.

### 1.2.2 C Arrays: "Close to the Metal"

C gives the programmer easy access to a fairly standard model of machine architecture (the "metal"). That makes it a language of choice for systems programmers, which is why you'd learn it in CPSC 213!

So, the *semantics*—the intended meaning—of arrays in C is fairly primitive.

- C Array Indexing: Syntactic Sugar for Juggling Addresses
  An `int` array variable in C can mostly be thought of as an `int` pointer variable that happens to point to the start of a block of memory with enough space for the whole array.

  Indeed, `a[i]` is actually *syntactic sugar*—surface syntax that's meant to be simpler ("sweeter") for the programmer than the underlying version. It is *desugared*—rewritten under the hood—to `*(a+i)`.[2] That underlying syntax means: take the memory location stored in `a`, add `i` to it[3], and look up what's at the resulting memory location.

  Is this really true? Well, if addition is commutative (i.e., `x+y` means the same as `y+x`), then `*(a+i)` is the same as `*(i+a)`, and `a[i]` should be the same as `i[a]`.

  Let's try it:

```
int array[5];
int i;
for (i = 0; i < 5; i++) {
  i[array] = i*i + 1;
}
printf("array length is %d; array[3] = %d\n",
       (sizeof(array)/sizeof(0[array])), 3[array]);
```

  Take a moment to think that through.

  This semantics ("arrays are just syntactic sugar for pointer arithmetic") leads naturally to many other features of C, including its lack of bounds checking[4].

  The same code using C++'s templated array type does *not* work:

```
std::array<int, 5> array;
for (int i = 0; i < 5; i++) {
  i[array] = i*i + 1;
}
std::cout << "array length is " << array.size() << "; array[3] = " << 3[array];
```

[2] In C11, `E1[E2]` is equivalent to `(*((E1)+(E2)))`.

[3] Actually the addition operator notices that one argument is a pointer and the other an `int` and multiplies the `int` by the size of the type pointed to by the pointer.

[4] Technically, per the C11 standard, it looks like bounds checking is implementation dependent. Practically, it's not there throughout the history of C.

The same goes for JavaScript.

- C Array Length: Statically Available, Dynamically Absent
  What about finding the array length. In fact, this shouldn't be possible. Why not?

  Consider: An array in C is a variable storing an address. Beginning at that address in memory is enough space allocated for that array to hold all of its elements. That's *all* that's in the variable and *all* that's in that block of memory.[5]

  In our example, with my compiler, here's what's stored in the variable and how much room is available:

```
int array[5];
printf("array contains: %d\n", (int)array); /* If you drop (int), this
                                               works the same.  That's
                                               because printf has no
                                               idea what type its
                                               arguments (after the
                                               first) are! */
printf("the amount of space available is: %d\n",
       (int)&array[5] - (int)&array[0]);
```

  Computers cannot magically produce information that they don't store. How am I getting the length?

  ANSWER: From the compiler. As the compiler compiles the code, *it* clearly knows that `array` is an array variable and how many elements it has, right? In the limited circumstance that the compiler has access to that information at the point in the code where we want the length of the array, we can get it.

  This won't work for *many* other examples, like this one:

```
int length(int argarray[]) {
  return sizeof(argarray)/sizeof(argarray[0]);
}
```

[5] In practice there might be some tagging information stored by the allocator near the array block, although not for statically allocated arrays. However, as a C programmer, what I've described is all you can generally count on.

```
/* Inside main: */
int array[5];
printf("length of array is: %d", length(array));
```

Why doesn't this work? Well, in order to work, the compiler must know which array `argarray` is. Does it know that? Can it know that?

This is our first example of the difference between *static* properties of a program—those associated with the program itself (i.e., the text of the program)—and *dynamic* properties—those associated with the running of the program. `argarray` could "be" several different arrays during the run of a program. So, the compiler cannot statically determine which array it is, much less how long it is! On the other hand, `array` is only ever one (shape of) array, and the compiler can determine its length statically.

Fortunately, both JavaScript and C++11 have *objects* and standard arrays we can use. So, their semantics for arrays are probably the same, right? :)

### 1.2.3 JavaScript Arrays

Note: This section builds from fascinating examples published in The Essence of JavaScript by Guha, Saftoiu, and Krishnamurthi.

JavaScript arrays are objects. To a programmer with, say, a Java (object-oriented programming) background, that's a natural statement. It produces a mental model akin to: "There's some class specifying functionality for arrays, and each array object is an instance of that class. The object presents an abstraction over the typical C-style fixed-length, mutable array."

- JavaScript Objects: Prototype-Based, Not Class-y; OOP, Not Monolithic
  JavaScript doesn't have classes. Inheritance is "prototype-based". Roughly speaking, that means that you create an object based on another object and until-and-unless you override some behaviour in a child object, it behaves like its parent object.

  This leads to some wacky behaviour. Unlike classes, objects can be altered at runtime (*dynamically*). Let's start extending our example..

```
var array = new Array(5);
for (var i = 0; i < 5; i++) {
```

```
  array[i] = i*i + 1;
}

// Array behaviour added at runtime!  (To all arrays, not just this one.)
Array.prototype.size = function() {
  var i = 0;
  while (this[i] != undefined) i++;
  return i;
}

console.log("array length is " + array.length +
            "; array.size() is " + array.size() +
            "; array[3] = " + array[3])
```

This takes a bit of getting used to, but you can probably imagine why you might want it. Maybe you design a function for "Book Title Case on Strings that Respects Whitespace" in Java. Wouldn't it be neat to just add it to the String type so that you can use it anywhere in your program on any String?[6]

Wouldn't it be horrible? Why?

(Think of a new colleague who suddenly runs across `String s = "the deathly hallows".toBookTitleCase()` in your codebase. How do they understand this code? How about a colleague with the same idea as you who also defines `toBookTitleCase`, but thinks "That" should be capitalized?)

So, Object-Oriented Programming is not really a monolithic idea understandable via learning an OOP language. That's why we instead study the toolkit a language designer uses to build languages.

- JavaScript Array Length: Three Semantic Surprises
  But hey, what about that new `size` function for arrays? It works the same as the already existing `length` property. Well, almost.

  You can delete elements from an array in JavaScript:

```
var array = new Array(5);
```

[6] Note that even if you could subclass `String`—which you can't because `String` is final—it wouldn't be the same thing. That would only affect strings of your subclass; we want all `Strings` to have a new behaviour.

```
for (var i = 0; i < 5; i++) {
  array[i] = i*i + 1;
}
console.log("array length is " + array.length +
            "; array[3] = " + array[3])

// Array behaviour added at runtime!  (To all arrays, not just this one.)
Array.prototype.size = function() {
  var i = 0;
  while (this[i] != undefined) i++;
  return i;
}

// Let's delete that element.  What do /you/ think a programmer should
// /expect/ to happen?
delete array[3]
console.log("array length is " + array.length +
            "; array.size() is " + array.size() +
            "; array[3] = " + array[3])
```

That's odd, but it gets stranger. You can assign to the `length` property:

```
var array = new Array(5);
for (var i = 0; i < 5; i++) {
  array[i] = i*i + 1;
}

// Array behaviour added at runtime!  (To all arrays, not just this one.)
Array.prototype.size = function() {
  var i = 0;
  while (this[i] != undefined) i++;
  return i;
}

console.log("array length is " + array.length +
            "; array.size() is " + array.size() +
            "; array[3] = " + array[3])

// Let's delete that element.  What do /you/ think a programmer should
```

```
// /expect/ to happen?
delete array[3]
console.log("array length is " + array.length +
            "; array.size() is " + array.size() +
            "; array[3] = " + array[3])

// Let's go crazy and /set/ the length to something different.  What
// do /you/ think a programmer should /expect/ to happen?
array.length = 8
console.log("array length is " + array.length +
            "; array.size() is " + array.size() +
            "; array[length-1] = " + array[array.length-1])

// Let's go crazier and set the length to something small.  What do
// /you/ think a programmer should /expect/ to happen?
array.length = 1
console.log("array length is " + array.length +
            "; array.size() is " + array.size() +
            "; array[4] = " + array[4])
```

Oh, and out-bounds-assignments in the array?

```
var array = new Array(5);
for (var i = 0; i < 5; i++) {
  array[i] = i*i + 1;
}

// Array behaviour added at runtime!  (To all arrays, not just this one.)
Array.prototype.size = function() {
  var i = 0;
  while (this[i] != undefined) i++;
  return i;
}

console.log("array length is " + array.length +
            "; array.size() is " + array.size() +
            "; array[3] = " + array[3])

// Let's delete that element.  What do /you/ think a programmer should
// /expect/ to happen?
```

```
delete array[3]
console.log("array length is " + array.length +
            "; array.size() is " + array.size() +
            "; array[3] = " + array[3])

// Let's go crazy and /set/ the length to something different.  What
// do /you/ think a programmer should /expect/ to happen?
array.length = 8
console.log("array length is " + array.length +
            "; array.size() is " + array.size() +
            "; array[length-1] = " + array[array.length-1])

// Let's go crazier and set the length to something small.  What do
// /you/ think a programmer should /expect/ to happen?
array.length = 1
console.log("array length is " + array.length +
            "; array.size() is " + array.size() +
            "; array[4] = " + array[4])

// Surely, we at least get an out-of-bounds error for assigning beyond the length
array[4] = "hello"
console.log("array length is " + array.length +
            "; array.size() is " + array.size() +
            "; array[4] = " + array[4])
```

That's just nuts. And, maybe it *is* nuts from a "programmer user interface" perspective.

From a semantic perspective, it flows from the design of JavaScript objects—to be flexible at runtime—and of arrays—to truly be objects under the hood, not something special.

- JavaScript Array Indexing: More Syntactic Sugar
  Indeed, arrays are not anything special. `array[1]` is *syntactic sugar* for a field access to a field named `1`, like `array.1` (or vice versa).[7] So, *of course* I can assign to an out-of-bounds element, just as I can add a new field to an object. Once I can do that, *of course* `length` needs to track those changes. And, perhaps, it's not a surprising design decision to let a `length` field that will already fluctuate be assignable as well.

[7] You're not allowed to write the `array.1` syntax, however.

And, that bracket syntactic sugar is by no means limited to arrays:

```
object = new Object()
object.foo = "bar"
object.bar = "foo"
delete object.bar
object["baz"] = "baz"
object[0] = "yes"
console.log("object's properties: " + Object.getOwnPropertyNames(object))

array = new Array()
array.foo = "bar"
array.bar = "foo"
array["baz"] = "baz"
array[0] = "yes"
array[1] = "no"
array[4] = "stop"
array[5] = "go"
console.log("array's properties: " + Object.getOwnPropertyNames(array))

// But, object still has no length property:
console.log("object's length property has the value: " + object.length)
```

Arrays in JavaScript initially seem familiar and intuitive to those raised on C, C++, or Java (or `vector` in Racket), but several language design choices made in JavaScript—and made very differently from C, C++, Java, and Racket—lead to an array that is radically different under the hood, with a real potential to bite the parochial programmer. That's why instead of studying languages, we study the toolkit a language designer uses to build languages.

### 1.2.4 C++ Arrays

Now, what could be weird about C++?

- C++ Array Length: Static or Dynamic?
  How about that `size` function? Is it like JavaScript's? Does it access a member variable?

  Well, no. In fact, C++11's `array` type is exposes access to C-style arrays, but with some of the advantages and trappings of C++. Under

the hood, it's: a contiguous block of memory long enough to store all the elements *and that's it*. Nonetheless, there is a function that to determine the size of the array.

You tell me: Is the size of the array determined *statically* or *dynamically*? How do you know?

Let's find out by writing our own `length` function:

```
template <size_t N> int length(std::array<int, N> a) {
  return N;
}

// Sorry, need to be explicit about the main function in this example.
int main(int argc, char * argv[]) {
  std::array<int, 5> array;
  std::cout << "length of array is: " << length(array) << std::endl;

  // Just for fun, a different array:
  std::array<int, 888> array2;
  std::cout << "length of array2 is: " << length(array2) << std::endl;
}
```

- C++ Array Length: "Dispatch" to the Right Function
  What happened?! To find out, let's compile the code only to assembly. You'll also want to "demangle" the result with `c++filt` as well. So, here's our code:

```
#include <iostream>
#include <array>

template <size_t N> int length(std::array<int, N> a) {
  return N;
}

int main(int argc, char * argv[]) {
  std::array<int, 5> array;
  std::cout << "length of array is: " << length(array) << std::endl;

  // Just for fun, a different array:
  std::array<int, 888> array2;
```

```
  std::cout << "length of array2 is: " << length(array2) << std::endl;
}
```

Here's how we "compile"[8] it:

```
g++ -std=c++0x -S lecture1.cc
c++filt < lecture1.s > lecture1.s.demangled
```

Here's a few parts of the assembly (those that mention "length"):

```
          ...
        call    int length<5u>(std::array<int, 5u>)
          ...
        call    int length<888u>(std::array<int, 888u>)
          ...
.globl int length<5u>(std::array<int, 5u>)
        .def    int length<5u>(std::array<int, 5u>);    .scl    2;      .type   3
int length<5u>(std::array<int, 5u>):
LFB1164:
        pushl   %ebp
LCFI11:
        movl    %esp, %ebp
LCFI12:
        movl    $5, %eax
        popl    %ebp
LCFI13:
        ret
          ...
.globl int length<888u>(std::array<int, 888u>)
        .def    int length<888u>(std::array<int, 888u>);        .scl    2;      .
int length<888u>(std::array<int, 888u>):
LFB1167:
        pushl   %ebp
LCFI14:
        movl    %esp, %ebp
LCFI15:
        movl    $888, %eax
```

[8] Actually, this really *is* compiling, whereas usually we compile, link, and assemble all at once.

```
        popl    %ebp
LCFI16:
        ret
          ...
```

At the top are two calls to `length`. To *two separate functions*, one for `5` and one for `888`. Later, we see the function definitions. Can you find where in the definitions the code "figures out" how long the array is?

Hint: the register named `eax` stores the return value of the function.

How did the compiler know *statically* what the length of the array parameter was?

Seems like cheating, doesn't it? :)

This is our first discussion of the issue of *dispatching*—choosing which version of a function to call. The solution here is the usual one for C: *static dispatch*. In C, most of the time, the compiler knows where the definition of a function resides given the name of the function. So, it replaces the function call with an assembly language instruction to jump to the implementation of the function.[9]

Can you guess whether the compiler knows where the function to be called is located in *dynamic dispatch*?

A good example of dynamic dispatch is the selection of the appropriate `toString` method in Java when we convert a particular object to a string. The `Object` class itself implements `toString`. However, if your object's class or any of its class's superclasses implements `toString`, then that will be called instead of `Object`'s version.

Therefore, without knowing how a function like the following is called, we cannot know which `toString` method it will invoke:

```
String toStringWrapper(Object o) {
  return o.toString();
}
```

We could call this on an `Integer`, a `String`, or any other type of object.

[9] C also lets you "take the address of" a function, which effectively yields the memory address where the code for that function begins. That's a value like any other address, and suddenly you can *sort-of* have higher-order functions like `map`.

This type of dynamic dispatch gives us one of the key powers that enable the OOP style. However, static dispatch tends to be more efficient. Sometimes, we can *statically* determine where some of these supposedly *dynamic* calls will go and therefore increase the efficiency of a call.[10]

Jumping up a level, that tension between *static* and *dynamic* will be one of the core themes of the course. What can/should we do statically? Dynamically? What different dynamic "flows" are there through a program? Which flows should we expose to the programmer, and how?

And one more time: That's why we study the toolkit a language designer uses to build languages, not the languages themselves.

## 1.3 How This Course Will Work

What we just did is *the* key learning goal in this course. We looked at different programming languages and understood deeply how and why they are similar and different. Understanding this allowed us to write code that exposes those differences and will in the future allow us to use the languages more effectively.

### 1.3.1 The "Hello, World! Bonjour, le Monde! ..." Approach to PL

One typical approach to a programming languages course is to quickly learn a bunch of languages—often one each from "classes" like object-oriented, scripting, procedural, functional, etc.—and use the opportunity to compare and contrast them. However, as our example above shows, this is an approach fraught with danger. Supposedly similar languages may harbor fundamentally different semantics. (After all, C++ and JavaScript are both object-oriented, right?) Furthermore, it's easy in this approach to focus on the surface features of the languages rather than the core constructs from which the languages are built.

### 1.3.2 The "Build It to Understand It" Approach to PL

Instead, we're going to iteratively *build* languages. We'll typically define the semantics of our language in words and by the interpreters we create to run

[10] The desire for static guarantees about which function will be called for a given function name is the major reason that `String` is `final` in Java; efficiency is one piece of this, but security is the major one!

them. We'll learn about new elements of programming languages as we add core constructs (and surface features) to our languages.

Often, a new core construct will require only small changes to the interpreter. You may be surprised at how profound an impact tiny changes in our interpreter can have. We will therefore also try to understand what powers these constructs give to our programmers and what dangers there may be in exposing them.

After all, quite a bit of programming language design is in taking power *away* from your programmer. Don't believe me? Try to imagine a more powerful language than assembly!

## 1.4 Logistics

### 1.4.1 Assignments: Programming and Conceptual

We'll have two types of assignments in the course: programming assignments (typically, investigating interpreters for various languages, particularly the "ParselTongue" language) and conceptual assignments.

- Programming Assignments: Marking, Groups, Demos (?), and Deadlines
  Programming assignments will be due every 1-2 weeks. All will be marked partially via automated testing and partially subject to review by the course staff.

  Your first project is posted on the course website under "assignments". It's to be completed *individually* subject to the very liberal collaboration policy described in the syllabus, **which you must read**!!!

  For later programming assignments, we hope to make two changes. First, pending replacement of a TA we lost to visa issues, we hope to do some submission review via in-person demos. Second, we plan to allow **and encourage** working in pairs for later programming assignments.

  Tentatively, we plan to have programming projects due Fridays at 8PM, with any demos scheduled as early as we can manage in the following week, but **let's discuss**!

- Conceptual Assignments: "Quizzes" Leading to Assignments; Timing
  Conceptual assignments are to be done individually, and for now we plan to keep it that way. Each conceptual assignment will come in two pieces.

Once a week at the start of class, we'll have a short multiple-choice quiz. You'll submit one copy of your answers on paper and grade the other copy yourself immediately.

Anything you miss will be due at the start of class one week later as a written assignment. In each case, your job will be to explain *why* the correct answer is correct and why the answer you chose is incorrect. (We're willing to entertain explanations of why our answer is wrong and yours is right... but I'd talk to us in office hours before submitting such an answer!)

Tentatively, we plan to have quizzes on Wednesdays at the start of class with homeworks due the following Tue at 8PM (via handin), but **let's discuss**!

Our goal with this setup is to make best use of both your resources for learning and ours for assessment. This will give you the chance to study in advance for the quiz and dramatically cut down the time you spend on conceptual assignments but to recover if the you have trouble with the topic. On the flipside, this means we mark and give feedback only on questions where you struggled, at least initially.

### 1.4.2 Reading: Do It

I will assume you've done the reading for lecture (and any associated quiz). If you haven't, trust me, lecture will be furiously fast and terribly confusing. See the website for the advance reading for a given day. (And yes, I understand many of you wouldn't have seen the "Read Chapter 1" instructions for the first course, but you should still go back and read it!)

### 1.4.3 Midterm Exams: Timing, Group Exam, and "Optional" Exams

Our midterm exams are open-notes, evening exams:

- Midterm #1: Tue, 8 Oct, 1900-2130, location TBD
- Midterm #2: Tue, 5 Nov, 1900-2130, location TBD

There are two problems with this:

1. How can we *require* you to come to an exam outside scheduled hours that may conflict with your hard constraints? Easy: we don't. Your

midterm exam mark (independently for each midterm) is the maximum of the midterm and your final exam mark. So, you can just skip the midterm. Do let us know if you are ill or have a hard conflicting constraint, however, and we will likely also back-calculate a midterm mark from your final mark via reasonable statistical manipulations. But, my advice: *take the midterm*!

2. That's a *long* time for a midterm exam. Is it really 2.5 hours long? No. We're going to shoot for 50 minutes. We'll then have a *group* exam almost immediately afterward. In a group exam, you'll work with 2-3 other students on the *same* exam you just took. Your personal grade will be the maximum of your individual exam mark and (0.75 * individual exam mark + 0.25 * group exam mark). Why? First, I think group exams are an awesome way to thrash out your ideas about the exam questions (i.e., really learn!). Second, quite a bit of research at UBC backs me up :)

### 1.4.4 Office Hours and Extended Office Hours: Your Time

Office hours are times when your course staff are available to work with you on whatever questions you might have. If there's time in a given session, you can even ask us about career choices, research, or how many kidneys Steve has. (We reserve the right not to answer personal questions.)

**WARNING**: just because you ask doesn't mean we'll answer. Our job is to help you answer your questions, not necessarily to just answer them outright.

You can find a schedule of times and locations for office hours right at the top of the syllabus. Some hours are still pending.

For the first two weeks, we're also holding extended office hours. Again, those are near the top of the syllabus and again, some are pending.

### 1.4.5 Tutorials: Targeted Exam Prep

The weekly Tuesday tutorials will be problem-working sessions. What problems? The TAs and I plan to rough out an exam question each week on the current material and design a tutorial problem based on it (but not the same problem!).

In other words: They're direct practice for likely exam questions.

The exams may have other questions, but they *will* each have at least two "tutorial" questions.

### 1.5 What have we learned today?

- Desugaring
  - Define the PL term "desugaring".
  - Recognize desugaring in an example.
- Semantics
  - Define the PL term "semantics".
  - Distinguish between syntactic and semantic issues in a programming language.
  - Illustrate how the semantics of syntactically similar constructs can differ.
- Static and Dynamic
  - Define the PL terms "static" and "dynamic"
  - Distinguish between static and dynamic properties of a program, given information about how information about those properties is determined (stored and/or computed).
  - Given information about whether a program property is static or dynamic, describe the impact on how it can be determined (stored and/or computed) and used.
- Dispatching
  - Define the PL term "dispatching" (with respect to function calls).

## 2 References


- ECMAScript Language Spec 5.1
- C11 Language Standard