Babylscript: Multilingual JavaScript

Ming-Yee Iu

March 12, 2012

1 Introduction

Current programming languages and their associated libraries are predominantly in English. This situation is beneficial in a professional environment because code and documentation can be easily shared between programmers in different countries due to their common use of English. In education and amateur contexts though, the dominance of English does cause problems. In non-English speaking countries, students cannot begin to learn programming until after they learn English, meaning these students will learn about programming at a later age than students in English speaking countries. The dominance of English also imposes a barrier preventing people who are not formally educated in programming and with a poor grasp of English from engaging in minor programming tasks such as scripting repetitive operations or writing simple business logic in web pages.

Although there is no technical reason that programming languages cannot be written using a non-English vocabulary, the widespread adoption of English programming languages produces a network effect that discourages the use of non-English programming languages. For example, a French programming language can only be designed, used, and maintained by French speakers. Since fewer people are able to evolve and maintain the language over time, the programming language will inevitably become less well-developed than English-based programming languages, to which the entire world programming community can contribute. Similarly, the software industry has a strong incentive to design their software libraries to have an English API since English is the lingua franca of the software community, so the libraries will be usable by more people and hence have a wider adoption if they are offered in English as opposed to another language. English-based programming languages can be viewed as more powerful and easier to use than non-English-based programming languages because there's a larger programming community available with which a programmer can share code samples and cookbook solutions with on online forums and elsewhere.

Babylscript is a programming language that is designed to be inherently multilingual. It is an extension of the JavaScript language and is designed to simultaneously support many vocabularies besides English. For example, Babylscript can be programmed using a French vocabulary, where the language keywords and library calls will all be in French (Figure 1). It can also be programmed using an English vocabulary, where the language keywords and library calls will all be in English. By having a common codebase

```
// English code
var win = new Window();
win.drawLine(3, 2, 20, 15, Color.RED);

// French code
---fr---
win.tracerLigne(2, 2, 19, 15, Couleur.ROUGE);
```

Figure 1: When programming in English, Babylscript objects export an English API. When programming in French, the same objects export a French API.

for all these different languages, the development resources of different linguistic groups can be pooled together into developing a single programming language instead of developing separate and distinct languages. Babylscript is more than simply a programming language that has been localized to use different vocabularies. It is designed with these goals in mind:

- Programmers should be able to program using their native vocabulary without being aware of the multi-lingual nature of Babylscript. Notably, the English version of Babylscript is normal JavaScript code.
- Programmers should be able to mix vocabularies in their code. Babylscript should allow programmers to borrow code samples written in French and mix it with code written in German, for example.
- The multilingual nature of the language should not be limited to only the core language and core libraries. Programmers should be able to extend their own code to expose a multilingual API

2 Programming Model

The Babylscript programming model takes the programming model of JavaScript and combines it with current practice in internationalizing computer programs. Babylscript requires changes to JavaScript's language, compiler, bytecode representation, virtual machine, and object model. These changes to the semantics of JavaScript are necessary because of JavaScript's dynamic nature and heavy use of reflection. Internationalization cannot be implemented through simple changes to the JavaScript syntax and isolated changes to the compiler.

2.1 Overview of Normal JavaScript

Normal JavaScript uses an object representation for its data structures. JavaScript objects are treated as collections of *properties*. A property is a data value and a name associated with that value (Figure 2). Because the names of properties can be arbitrary strings, JavaScript objects are often treated as associative arrays, a data structure that maps strings to data. The members of an object can be accessed using `object.x` where `x` is the name of a property. The values of a property can be numbers, strings, references to other objects, or even functions. Object methods are implemented as object properties that refer to functions.

The JavaScript object model is based on prototypes instead of classes. Unlike in a class-based system, a programmer can add and remove the properties of individual JavaScript objects at runtime. JavaScript supports inheritance by allowing objects to have a link to a *prototype* object. When a program tries to access an object's property with a certain name, JavaScript will first look to see if there is a property with

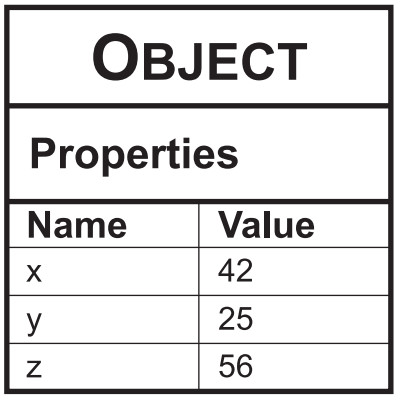

Figure 2: JavaScript objects are represented as collections of properties

the correct name in the object itself. If no such property is found, JavaScript will then look in the prototype to see if the prototype has such a property (Figure 3). Prototype objects are normal JavaScript objects, and they may have their own prototype objects. When performing a lookup for a property, JavaScript will traverse this prototype chain as necessary to find the desired prototype.

JavaScript also models its global state as an object. Global variables are stored as properties in an object usually called `window`. For example, a global variable named `config` exists as the property `window.config` in the `window` object. Programmers can create, delete, and enumerate the global variables of a JavaScript program by inspecting this `window` object.

2.2 Overview of Typical Internationalization Approaches

Most computer applications typically use the same approach for handling internationalization. All the text used in a program are gathered to-

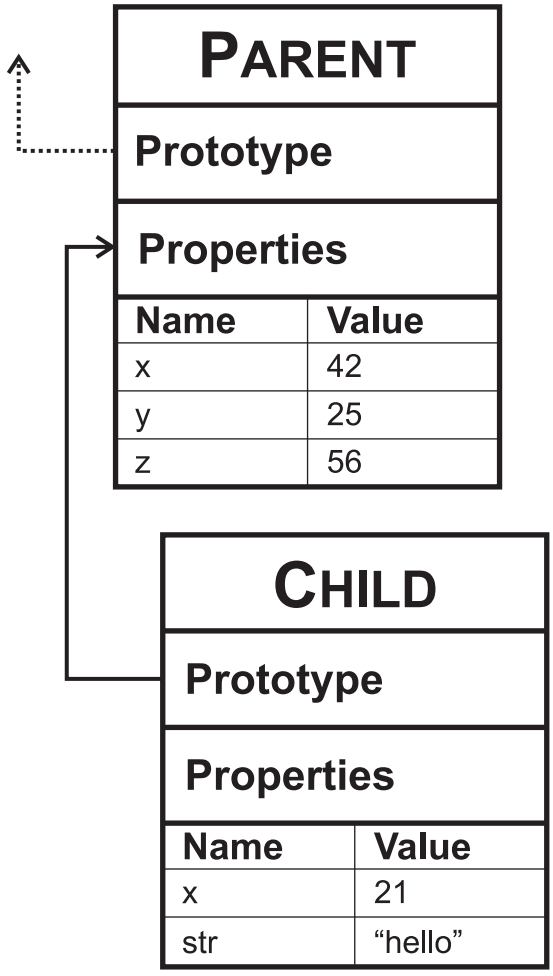

Figure 3: When looking up a property in an object, JavaScript first checks to see if the property exists in the object itself. If it cannot find the property, it will check the object's prototype to see if it has the desired property.

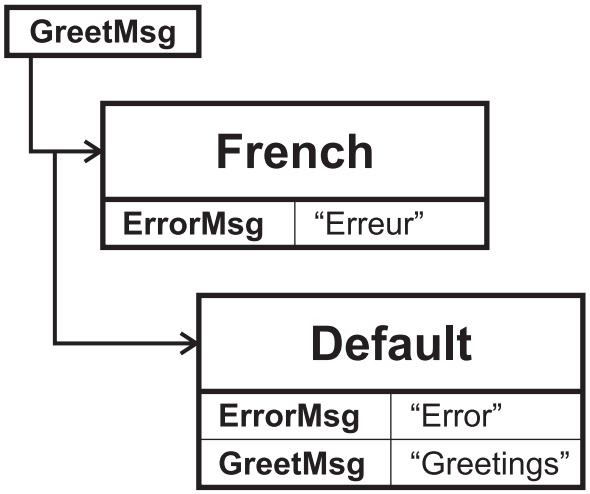

Figure 4: When looking for the text for the message **GreetMsg**, an internationalized application will first look at the lookup table of messages available in the local language of French. Since a translation of that message is not available, the application will need to use the text from the lookup table of untranslated messages.

gether in a file. This file is organized as a lookup table. When a program needs to display a message, it searches this lookup table for the text of the message, and then shows the text.

To translate an application, the file of messages is copied, and all the text translated. When an application displays a message, it will first search this translated file for the message. Due to incomplete translations, a translation might not be available in the translated lookup table. In those cases, the application can fall back to the original lookup table of messages written in the original default language (Figure 4).

2.3 Babylscript Programming Model

Babylscript allows programmers to use a mix of different vocabularies in their code. An object might be created by code written in English, manipulated by code written in French, and then given as a parameter to a library written in German. Depending on the context in which an object is used, the object should export an API using the appropriate vocabulary. Objects should expose English fields and methods when being manipulated by code written in English, and they should expose French fields and methods when being manipulated by code written in French.

Babylscript extends the JavaScript object model by allowing properties to have different names depending on the context. Programmers can provide translations of the names for their objects' properties, and Babylscript will choose the appropriate name depending on the context of the code being used to manipulate the object. Since programmers might not provide translations for all of the names in their programs, Babylscript also supports the concept of a *default name* that can be used to refer to a property if an appropriate translation for the name is not available.

Similar to normal JavaScript, Babylscript objects are collections of properties consisting of values and their associated default names. Unlike normal JavaScript, Babylscript objects also contain special lookup tables that contain translations of the names of properties. These lookup tables hold 1:1 bidirectional mappings between a property's default name and the translation of that name in a particular vocabulary. For example, Figure 5 shows an object with three properties: `show`, `x`, and `y`. The object also has two lookup tables that hold translations of the

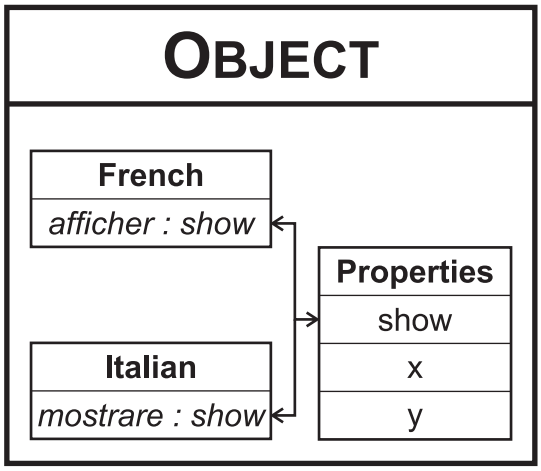

Figure 5: In addition to having a collection of properties, Babylscript objects also have contain mappings between untranslated default property names and translations for those names

property named `show` for French and Italian vocabularies.

When some code tries to access an object's property with a certain name, Babylscript will first search the object's lookup table associated with the current vocabulary. If the name exists in the lookup table, Babylscript will find the original untranslated default name of the property, and return that property. If no such name exists in the lookup table, Babylscript will see if a property with a matching default name exists. For example, if some code written in French were to access a property `afficher` of the object shown in Figure 5, Babylscript would first look in the French lookup table. Since `afficher` does exist in the lookup table, Babylscript will be able to follow the mapping and return the associated `show` property. If some code written in French were to access a property `x`, Babylscript will not be able to find an appropriate entry in the French lookup table. As such, it will search the list of properties and return the `x` property directly. Following these rules, if some code written in French were to access a property `show`, Babylscript will directly return the `show` property. One interesting feature of JavaScript is that Babylscript's extensions to the JavaScript object model automatically apply to global variables as well. Since JavaScript uses an object to store global variables, adding support for translated names to the object model results in global variables supporting translated names as well.

If a property with a certain name is accessed in an object and that object has a prototype, then the procedure for finding the property is slightly more complicated. Babylscript will first look in the lookup tables of the object and its prototype chain to see if the name is a translation or not. If no translation is found, Babylscript will return at the original object and traverse its prototype chain looking for a property with a matching default name. If a translation is found, Babylscript will also return to the original object and search the prototype chain but use the original default name of the property and not the supplied translated name.

JavaScript allows programs to iterate over the names of properties in an object, so Babylscript must support this operation as well. In order to enumerate the names, Babylscript walks over the default names of the object's properties. If the default name has a translated name in the current vocabulary, Babylscript will give the translated name when iterating over the names; otherwise, Babylscript will return the default name. It is possible for there to be a name collision when an object has a default name and a translated name that are the same but where the translated name does not map to that default name.

```
object['fr':'alerter'] = 'alert';
```

Figure 6: A definition of a mapping from the French name "alerter" to the default name "alert"

In these situations, the behavior of Babylscript is undefined.

In Babylscript, the lookup tables of translated property names are manipulated separately from normal properties. Programmers must explicitly create, maintain, and delete these translations. If some code written in French create a property in an object, Babylscript does not create any implicit entries in the French lookup tables. If a property is deleted from an object, the associated translations of the property name are not removed.

3 Syntax

As mentioned earlier, Babylscript is derived from JavaScript, and the syntax of the English version of Babylscript is identical to normal JavaScript but with a few extensions for multilingual support.

One of these extensions is the ability to specify alternate translated names for properties. Figure 6 shows how a new translated name is defined. The syntax borrows from JavaScript's syntax for associative arrays. Instead of simply specifying a name and a value to bind to that name, the programmer specifies a language, a translated name, and the default name to be bound to the name. All operands are strings expressions.

When a programmer wants to write code using a non-English vocabulary, Babylscript allows the programmer to switch *language modes*. In the French language mode, for example, variables names and keywords are entirely in French.

```
function hello()
{
    alert('Hello world!');
    ---fr---
    alerter(«Hello world!»);
}
```

Figure 7: Sample syntax for a function that prints "hello world" twice

A programmer can switch between different language modes by typing three minus signs, the name of the desired language vocabulary, and then another three minus signs (Figure 7). This command is the same in all the language modes.

Once Babylscript is configured for a certain mode, programmers can write programs using that language's vocabulary without being aware of Babylscript's support for other vocabularies. Although different language modes have different keywords and variable names, the different modes share the same basic *grammar*. The terminal symbols of the grammar do resolve to different keywords and names depending on the mode, but the basic structure remains the same. This sharing of the grammar greatly simplifies the task of localizing Babylscript for other vocabularies while not particularly disadvantaging any particular translation of Babylscript since the JavaScript grammar is already sufficiently language neutral.

Since the same grammar parser is used to handle all the different language vocabularies supported by Babylscript, customizations needed to handle specific vocabularies can only appear in the scanner stage of Babylscript. This restriction can be cumbersome at times, such as when dealing with tokens that have different meanings in different languages. For example, many countries use a comma as a decimal separator. This

```
---fr---
var a = Math.max( 1,0 , 2,0 , 3,0 );
var b = Math.max( 1,0 ; 2,0 ; 3,0 );
```

Figure 8: When commas are used as a decimal separator, it can lead to code that is theoretically unambiguous but difficult for humans to read. Babylscript accepts semi-colons as an alternative to the semi-colon to improve the readability of the code

can potentially lead to code that is confusing for people to read (Figure 8) because JavaScript also uses commas for the argument lists of functions. Babylscript solves this problem by adjusting the JavaScript grammar to accept either commas or semi-colons between function arguments. Another possible solution would be to adjust the scanner to shift the meanings of symbols, so that commas would be treated as periods, periods as semi-colons, and semi-colons as commas.

4 System Architecture

Babylscript is implemented as a modification to the Mozilla Rhino JavaScript interpreter. Rhino is implemented in Java and is included in current versions of the Java virtual machine. The interpreter translates JavaScript code into JavaScript bytecodes. The interpreter can then execute this bytecode or optionally compile this code to Java bytecode for direct execution by the Java virtual machine. This compilation is disabled in our implementation.

Figure 9 gives an overview of the modifications needed to Rhino in order to implement Babylscript. Changes were made to five of the stages of the Rhino JavaScript implementation: the scanner, parser, bytecode generator, interpreter, and libraries.

4.1 Scanner

Since the different language modes share the same grammar, most of the handling of different vocabularies must be done by the scanner. The scanner uses the language-specific rules of the current language mode when breaking a stream of text into keywords and other tokens. The command for changing language modes is handled directly by the scanner. The scanner switches its language mode internally without outputting anything to the parser.

4.2 Parser

Although the grammar of the different language modes of Babylscript are identical, the lookup of variable names and object fields is dependent on the language mode. As a result, all of the tokens passed from the scanner to the parser are tagged with the language mode under which the token occurred. The parser propagates this contextual information when it builds its syntax tree. The parser is also extended with support for new operations for manipulating translated names.

4.3 Bytecode Generator

Babylscript extends Rhino's bytecode with new instructions for manipulating translated names and with a language register. The language register stores the current language mode as a string and is used when performing lookups of translated names in objects. Babylscript's bytecode generator uses the language tags in the parse tree to generate bytecode that updates this register before any operation that accesses an object's properties.

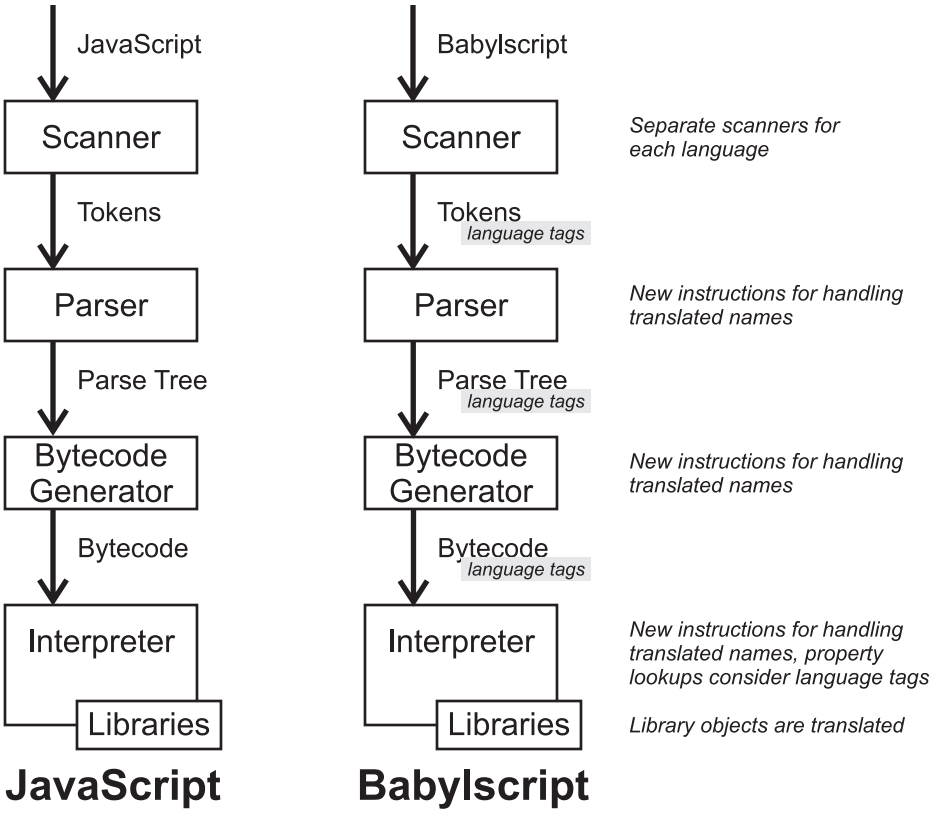

Figure 9: Building Babylscript from Rhino requires the creation of different scanners for different vocabularies, the addition of new instructions for manipulating translated names, and the tracking of language modes through the toolchain

4.4 Interpreter

As described in the Section 2 on the Babylscript programming model, Babylscript extends the JavaScript interpreter by taking language into account when accessing properties.

4.5 Libraries

Fortunately, the JavaScript standard library is extremely small. In Babylscript, the JavaScript libraries are extended with translations into other languages. Babylscript's support for translated names is a sufficient to allow objects and their members to have different names in different languages.

One unfortunate aspect of the JavaScript standard library though is that it is very US-centric. Although Babylscript provides translations for method and object names, much of the JavaScript standard libraries inherently use a US localization in handling number formats and dates. Although the standard library was designed so that it could be later extended with internationalization support, this support was never filled in until recently during the standardization process for ECMAScript 6. For example, number and date objects all have a toLocaleString() method that potentially allow programmers to create localized string representations of an object. But no attempts have been made to specify the parameters to this method until recently. Similarly, the functions for parsing numbers by default only accept numbers in US-specific formats. These parsing functions potentially accept extra parameters that allow them to parse numbers in other formats, but these parameters have also been left unspecified until recently.

Additionally, JavaScript makes liberal use of implicit conversions between numbers and strings, and all of these conversions expect US number formats. In order to retain compatibility with previous versions of JavaScript, future versions of JavaScript will likely retain this behavior. For example, JavaScript objects are often used as associative arrays. When one indexes into an associative array using a number, the number is converted to a string first. If the conversion of numbers to strings were dependent on the current locale, then these lookup operations might fail for some users because the string representation would differ depending on the locale.

Although other programming languages do take the locale into account even for implicit conversions, these languages usually impose this requirement early in their history, so that programmers could adapt their programs to handle different locales. Despite this, beginner programmers using these programming languages still often have problems writing code for reading text configuration files (e.g. XML). These files might contain numbers in a US-format, but if the configuration file is read using a locale with a different number format, the number conversions will fail. Given that JavaScript has not historically supported allowing programmers to specify that number conversions should use a culturally neutral or other locale, programmers have not been able to write their programs in a way that could handle implicit number conversions being locale-specific.

Having number and date conversions be US-centric would be unacceptable for Babylscript. Babylscript would then no longer treat English, French, Chinese, and other languages equally. For example, English users would be able to concatenate strings and numbers together easily using a plus operator while French users would ei-

ther need to use special conversion functions or be content with using numbers formatted using a US number format. Such a situation would be especially problematic if Babylscript were to be used in educational settings. Non-English speakers would need a deep understanding of programming before they could do something simple like display a mix of text and numbers on the screen. Babylscript needs to balance the goal of parity among different cultures with compatibility with JavaScript.

To balance these two goals, Babylscript restricts itself to only supporting the most common number formats. Although the world does have a wide variety of ways for representing numerals, in practice, many of these numerals are archaic or not commonly used in mathematics or not supported by electronic calculators. Number systems that don't have a symbol for zero or that don't support the concept of negative numbers will never be fully supported by modern programming languages. By supporting only a restricted set of number formats, Babylscript's functions for string to number conversions can simultaneously support all of these formats without depending on a particular locale. For example, Babylscript treats both commas and periods as a decimal separators in numbers. Babylscript's number to string conversions are locale-specific, but they default to outputting numbers in a generic style that's easier to parse (e.g. no grouping by thousands with thousands separators etc.).

5 Future Work

Currently, we see three main areas where further research into multilingual programming can be undertaken:

5.1 Performance

Babylscript has more runtime overhead than normal JavaScript. Every property access is slower because Babylscript must lookup every name twice: once in the list of translated names and again in the list of object properties. Existing techniques for optimizing dynamic dispatch in JavaScript can likely be used to reduce this overhead, but these techniques require major changes to the JavaScript runtime engine. Babylscript is also slower than normal JavaScript during initialization of programs because these programs must currently initialize the different translations for object properties, which can be considerable if a library supports many different languages. Support for lazily initializing these programs may partially reduce this overhead.

5.2 Multilingual Statically Typed Languages

Since statically typed languages have much more reliable type information than a language like JavaScript, one interesting direction would be to build a Babylscript-type language on top of Java instead of JavaScript. This type information might make it possible to build multilingual support entirely inside the compiler, without requiring any changes to the virtual machine. If all handling of translated names were done at compile-time instead of at runtime, this multilingual Java would have no performance penalty over normal Java, unlike Babylscript. It would also be possible to build much more complicated schemes for handling translated names. For example, a French programmer using a Chinese library might prefer a French interface for the library, but be willing to settle for an English or German interface if a French one isn't available

rather than having to resort to the default Chinese interface.

Building a Babylscript-type language on top of Java would require the design of new language mechanisms that aren't required in Babylscript though. For example, in Java, libraries are generally shipped to programmers in a compiled form that is not intended to be modified. Programmers, though, may want to modify a library by adding a translation to it, so a special mechanism for patching translations into sealed libraries would be required. Another example would be a special mechanism for handling method overrides. In Java, libraries are often extended by subclassing an object and overriding existing methods. A programmer might use a translated name when overriding a method, so a mechanism is needed to compile this method to use the default name instead. This situation can potentially also occur in JavaScript, but is ignored by Babylscript because the existence of closures and anonymous functions in JavaScript means that JavaScript libraries are generally extended through the use of callback functions and not through JavaScript's cumbersome support for inheritance.

5.3 Automated Translations

Although Babylscript does facilitate the sharing of code written using different vocabularies, the holy grail of multilingual programming is support for the automated translation of code from one vocabulary to another such as from Spanish to French. Full automated translation of program code and its comments is probably impractical given the difficulty of the automated machine translation task in general. Fortunately, in Babylscript, most libraries have translations available for function names etc., so some sort of limited translation should be possible. Unfortunately, Babylscript is based on JavaScript, and since JavaScript is a very dynamic language, a lot of interprocedural analysis would be needed to generate type information, so the translation would be very imperfect and may not result in executable code. A statically-typed version of Babylscript could potentially allow for limited but robust automated translation of code between two different vocabularies though.