PEJS

Python Executed in JavaScript

Jesper Jakobsen
20052904
u052904@daimi.au.dk

Rune Fogh
20052251
rfogh@daimi.au.dk

Ubbe Welling
20052275
ubbe@daimi.au.dk

Contents

1 Introduction

This report is the final handin in the course ”Design of Virtual Machines for Object Oriented Languages” taught at Department of Computer Science at Aarhus University during the fall semester of 2008. The course has been taught by Lars Bak and Kasper Lund, assisted by Mathias Schwarz.

The report describes the design and implementation of a virtual machine for the programming language Python done in JavaScript. The purpose of this report is to convey a description of our work process, the status of the project and the results obtained so far. The deadline for this report was December 17th 2008.

The project source code is available at `http://code.google.com/p/pejs`.

2 Motivation

Virtual machines for JavaScript exist in all modern browsers and since browsers exist on most operating systems JavaScript is virtually platform independent. So if we could build a virtual machine in JavaScript that executes a Python program we can make Python platform independent.

Further, the growing use of JavaScript on the internet has pushed the need for an efficient virtual machine. The work by Google[1] and Mozilla[2] has made it interesting to investigate if it is viable to build a virtual machine on top of a JavaScript virtual machine, which is historically known for having poor performance.

3 Goals

Here we briefly state our initial goals and milestones for the project.

Our main goal was:

In JavaScript, implement a VM for a subset of the Python language.

This was broken down into smaller goals, or milestones, as follows:

1. Get a simple interpreter operating on a stack up and running
2. Implement basic constructs: Jumps and loops
3. Decide on object representation and heap layout
4. Implement classes and objects
5. Implement simple garbage collector if necessary
6. Handle calls to Python library somehow
7. Do performance evaluation and optimizations

As we only had around 8 weeks for the project, we decided that points 1-4 were required while 5 and 6 could be omitted if necessary. Performance evaluation was naturally a must, and optimization was also something we definitely aimed at working with.

[1]http://code.google.com/p/v8

[2]http://www.mozilla.org/js/spidermonkey

4 Implementation

4.1 Overview

The project consists of two parts, compiling the Python code to Python opcodes in a format we can interpret, and interpreting the opcodes in JavaScript. The focus of this course is to make a VM, so naturally we focus on the interpretation part rather than the compilation part. In order to save time we use the Python compiler to get the opcodes and other relevant information and output it as needed.

The opcode format is formed as a JavaScript object we call a code object. This is illustrated by figure 1.

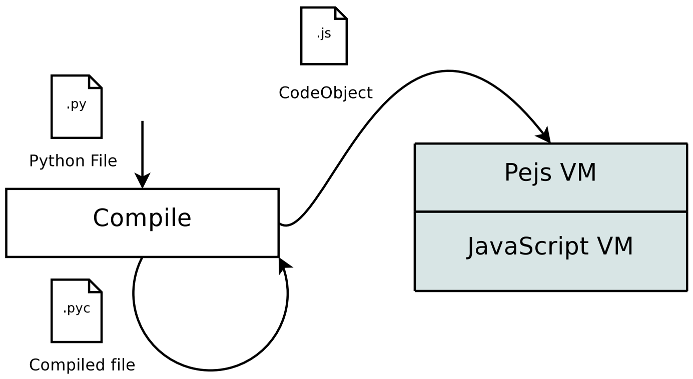


Figure 1: Overview of PEJS

We compile the Python file with `tools/compile.py` which outputs a JavaScript file with the code object. To illustrate this here is an example of a py file and the corresponding code object.

Algorithm 1 Helloworld.py

```
class Greet:
  def printGreeting(self):
    print "Hello world!"
Greet().printGreeting()
```

Algorithm 2 Helloworld.js

```
//This file was automatically created with compiler.py

var Helloworld = {
  co_name: "?",
  co_argcount: 0,
  co_nlocals: 0,
  co_varnames: ["Greet"],
  co_code: [100,0,0,102,0,0,100,0,1,132,0,0,131,0,0,89,90,0,0,101,0,0,
           131,0,0,105,0,1,131,0,0,1,100,0,2,83],
  co_consts: ["Greet", "CODEOBJ: Helloworld_Greet", "None"],
  co_names: ["Greet", "printGreeting"],
  co_locals: [],
  toString: function() { return "CodeObject:Helloworld"}
};

var Helloworld_Greet = {
  co_name: "Greet",
  co_argcount: 0,
  co_nlocals: 0,
  co_varnames: [],
  co_code: [116,0,0,90,0,1,100,0,1,132,0,0,90,0,2,82,83],
  co_consts: ["None", "CODEOBJ: Helloworld_Greet_printGreeting"],
  co_names: ["__name__", "__module__", "printGreeting"],
  co_locals: [],
  toString: function() { return "CodeObject:Helloworld_Greet"}
};

var Helloworld_Greet_printGreeting = {
  co_name: "printGreeting",
  co_argcount: 1,
  co_nlocals: 1,
  co_varnames: ["self"],
  co_code: [100,0,1,71,72,100,0,0,83],
  co_consts: ["None", "Hello world!"],
  co_names: [],
  co_locals: [],
  toString: function() { return "CodeObject:Helloworld_Greet_printGreeting"}
};
```

In order to run the Python code in a browser you need to import the interpreter (`src/interpreter.js`), the standard library (`src/lib/stdlib.js`) and the code object file, in this example `Helloworld.js`. In an html file you then need to execute the `interpret` method with the name of the module as a string:

```
(new PEJS()).interpret("Helloworld");
```

See appendix B for more details on running PEJS.

The above implies that the author of the website needs a Python compiler to generate code objects. That does not apply to the users of the website. It is possible to make a Python compiler in JavaScript which could be imported in the html file. That would allow Python directly embedded in the html. This is not in the scope of the course and therefore not implemented.

4.2 Compiler

As our main focus of this project has been implementing a VM, the compiler has been built quick and dirty in Python and actually just uses Pythons own compiler. The main purpose of the compiler is to compile Python source code to Python byte code and present it in a JavaScript format. We wanted to make our VM run Python byte code, so in that respect, the format was settled, but at the same time, we wanted to experiment with the way we represented the byte code in JavaScript.

4.3 Code objects

Python has a notion of code objects as containers of code. We have code objects for classes, functions, methods and the main program, in Python known as a module. The compiler compiles and decompiles the Python source code and generates the proper JavaScript code (figure 1). Besides containing the actual byte codes, code objects also contain several properties discovered during compilation. At first in the project, we represented all properties from Python code objects in our JavaScript representation, but since we haven't seen any use for some of them so far, we decided to remove the unused properties. The properties being used are described in table 1.

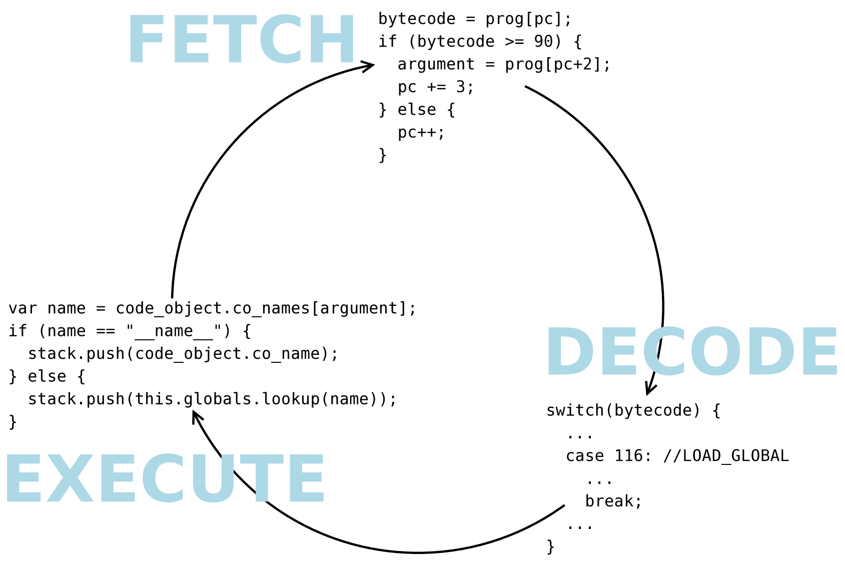
Property	Explanation
`co_name`	Name of the class or function
`co_argcount`	Number of arguments expected by function
`co_nlocals`	Number of local variables used in the code object
`co_varnames`	Array of local variable names
`co_code`	Array of byte codes, represented as integers
`co_consts`	Array of constants such as string, integers and other code objects
`co_names`	Array of names of functions, classes and/or properties
`co_locals`	Empty array used to store local variable values

Table 1: Our Javascript representation of Python code objects.

The byte codes have been represented in two different ways during the project. At first, we represented each byte code as an array in order to carry a lot of debugging information along with the byte codes. Later in the project, when we settled on the basic structure and most of our VM worked as in the current state, we changed this to a more effective representation, where each byte code is one or three entries in an array instead of each instruction being an array by itself. The optimization benefits of this change is discussed later in section 5.2.2.

4.4 Interpreter

The interpreter is designed as a simple fetch-decode-execute cycle. The main structure of the interpreter is a while-loop with a switch-case that executes code objects. As shown in the example in figure 2, the interpreter maintains a program counter (`pc`), which is an array index in the code objects `co_code` property. The byte code indexed by `pc` is read out to the `bytecode` variable. In Python, all byte codes of value 90 or larger has an argument as well, so if this is the case, the argument is read out as well, and `pc` incremented correspondingly. A switch tests the current bytecode to get the correct case, and the contents of the case is executed.



```
bytecode = prog[pc];
if (bytecode >= 90) {
  argument = prog[pc+2];
  pc += 3;
} else {
  pc++;
}
```

```
switch(bytecode) {
  ...
  case 116: //LOAD_GLOBAL
    ...
    break;
  ...
}
```

```
var name = code_object.co_names[argument];
if (name == "__name__") {
  stack.push(code_object.co_name);
} else {
  stack.push(this.globals.lookup(name));
}
```

Figure 2: Fetch-decode-execute cycle for byte code `116`, `LOAD_GLOBAL`

The fetch-decode-execute cycle is initiated when a new function is called and terminates with the byte code `RETURN_VALUE` or when an exception is raised without being catched in the current cycle. At the moment, exceptions are not fully supported. See more in section 5.1.7.

At the moment, we pay the full penalty of the switch-case, but we have some suggestions on how to improve performance of the cycle. These suggestions can be found in section 6.

4.5 Static structure

The virtual machine is structured as shown in figure 3. The interpreter has been implemented on the `prototype` object of the `PEJS` constructor. Another constructor is also implemented on the `prototype` object, namely `Globals`, which is described in more details in section 4.6.2. Furthermore, the `types` ”namespace” (implemented as an object) on the `prototype` object contains the constructors for all built-in types.

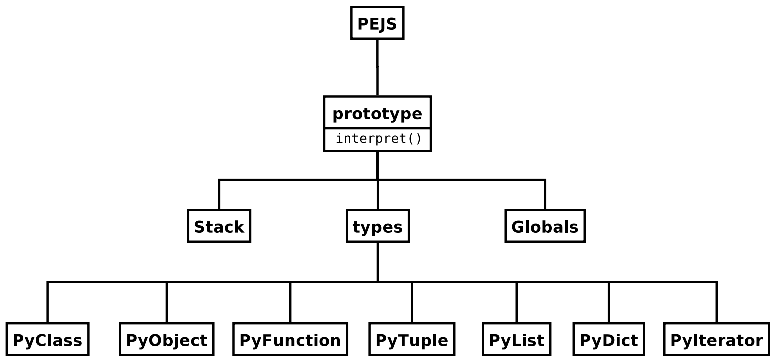


Figure 3: Static structure of `interpret.js`

4.6 Memory model

4.6.1 Stack

The stack is represented by a separate stack frame for each code object that is run. Each frame is simple: A regular JavaScript array where we utilize the `push` and `pop`-methods provided, and add our own `peek()`. Representing the stack this way means that we do not have to keep track of stack pointers at all. It also means that JavaScript can automatically garbage collect stack entries that have been popped, as well as entire frames that are no longer in use.

4.6.2 Local and global variables

Local variables are stored in the respective code object's `co_varnames` and `co_locals`. The compiler resolves most of the names and adds them to `co_varnames`, and on runtime, the corresponding values are stored at the same index in `co_locals`. When a function is called, a new stack frame is created, with space allocated for the local variables.

Global variables are stored in the `Globals` object, which consists of an array of arrays and some convenience methods. The first inner array is a special array reserved for new global variables, while the other inner arrays are added on runtime as direct references to the outermost code objects's `co_varnames` and `co_locals`. The reason for this odd construction is the Python compilers lacking ability to resolve whether a variable is local or global at compile-time, so the best working solution we have come up with so far is the one just outlined.

Lookup of unresolved variables, implemented as the byte code `LOAD_NAME`, is done by first looking through `co_varnames` and secondly through the global variables. Each of these lookups takes time linear in the number of local and global variables, respectively.

4.6.3 Objects

Python does, not surprisingly, use classes, objects and functions. We have choosen to implement these elements as JavaScript objects. The properties and relations of these are given in figure 4.

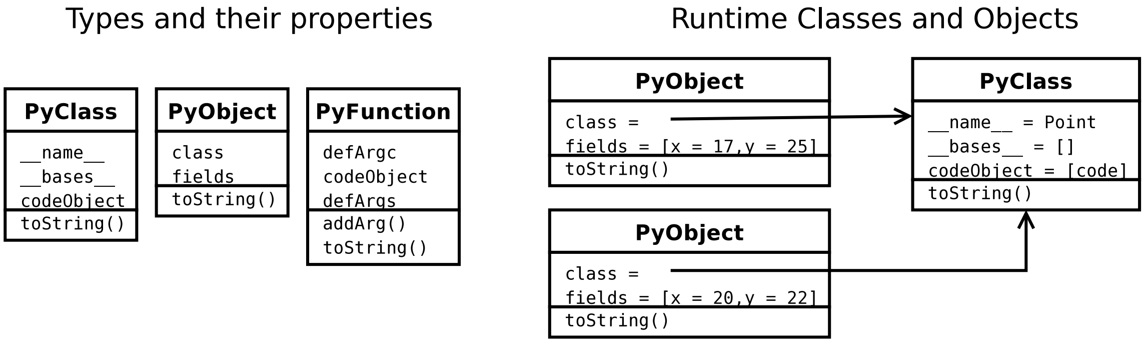


Figure 4: Representation of Python objecs and classes

4.6.4 Tuples, lists and dictionaries

Python has three basic collection types, namely tuples, lists and dictionaries. Lists are mutable collections of values, corresponding to JavaScript arrays, while tuples are immutable lists. Dictionaries are a mutable collection of key-value pairs, similar to a Javascript object. The obvious approach was therefore to represent dictionaries as JavaScript objects and lists and tuples as JavaScript arrays. The three collection types are represented as PyTuple, PyList and PyDict, which each has a `store` property, that contains an array or an object, respectively.

Methods on lists and dictionaries are implemented in JavaScript only, which was chosen because it was the simplest approach at the time, although they possibly to a greater extent could be part of the standard library which is mostly implemented in Python.

4.7 Library

We have started an implementation of the Python library in order to support some of the most basic functionality. While we have not implemented much, the idea was just as much to see if we could get a library working at all.

The library is primarily implemented in Python which seemed most elegant, but we still had to establish some way of calling into JavaScript which is done as in the following example:

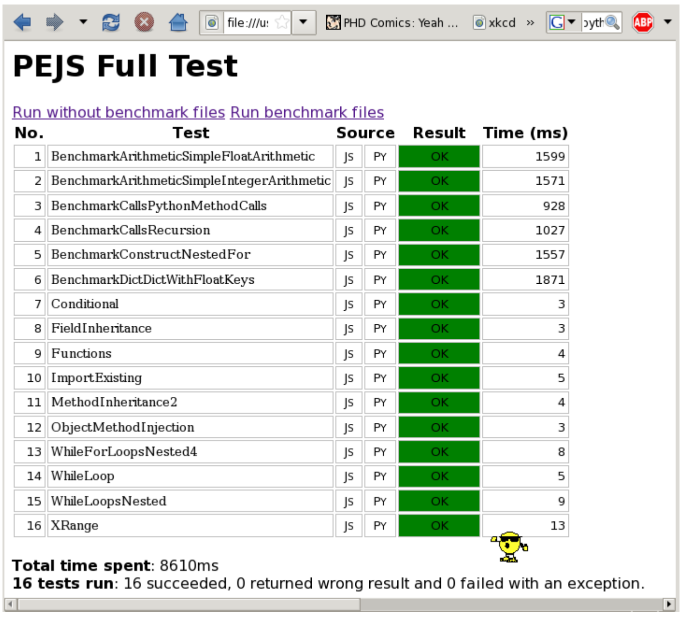
```
exec "new PEJS.prototype.types.PyTuple("+list+");" in "JavaScript", "result"
return result
```

In the EXEC_STMT case in the interpreter, we test if the second parameter is "JavaScript", in which case we evaluate the given statement and store the result in the named variable, which is then accessible in Python. This clearly enables arbitrary extensions of the library, as it is always possible to switch to JavaScript implementation when Python is not sufficient. It is also easy to add new implementation: One just writes the code in the right place, and everything works automatically.

The library .py-files are translated and the generated JavaScript file is then referenced in the html, just like the interpreter itself. Writing "import <module>" then works exactly as it should. The standard library is imported automatically, as it should be.

4.8 Test framework

During the development of our VM, it became obvious that we needed to establish some kind of systematic testing in order to make sure that we didn't break existing functionality when refactoring and adding new features. Thus, we set up a few scripts to automatically translate tests written in Python into our JavaScript representation. We could then benefit from the fact that our VM runs in a browser in that we pretty quickly had a html interface (`test/test.html`) working in which we could run all tests and also see detailed runs of single tests. We used the html interface for running benchmarks as well.



PEJS Full Test

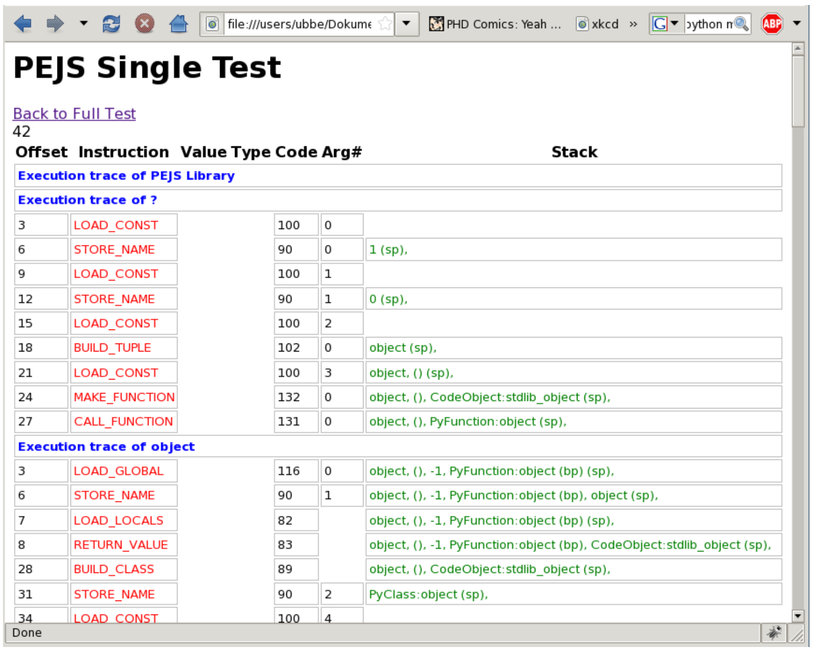
Run without benchmark files Run benchmark files

No.	Test	Source		Result	Time (ms)
1	BenchmarkArithmeticSimpleFloatArithmetic	JS	PY	OK	1599
2	BenchmarkArithmeticSimpleIntegerArithmetic	JS	PY	OK	1571
3	BenchmarkCallsPythonMethodCalls	JS	PY	OK	928
4	BenchmarkCallsRecursion	JS	PY	OK	1027
5	BenchmarkConstructNestedFor	JS	PY	OK	1557
6	BenchmarkDictDictWithFloatKeys	JS	PY	OK	1871
7	Conditional	JS	PY	OK	3
8	FieldInheritance	JS	PY	OK	3
9	Functions	JS	PY	OK	4
10	ImportExisting	JS	PY	OK	5
11	MethodInheritance2	JS	PY	OK	4
12	ObjectMethodInjection	JS	PY	OK	3
13	WhileForLoopsNested4	JS	PY	OK	8
14	WhileLoop	JS	PY	OK	5
15	WhileLoopsNested	JS	PY	OK	9
16	XRange	JS	PY	OK	13

Total time spent: 8610ms
16 tests run: 16 succeeded, 0 returned wrong result and 0 failed with an exception.

Figure 5: Screenshot of full test.

In figure 5, each test can be clicked in order to run a trace of that particular test. In figure 6, each blue headline indicates that control has been transferred to another code object. The stack grows to the right.



PEJS Single Test

Back to Full Test
42

Offset	Instruction	Value	Type Code	Arg#	Stack
Execution trace of PEJS Library					
Execution trace of ?					
3	LOAD_CONST		100	0	
6	STORE_NAME		90	0	1 (sp),
9	LOAD_CONST		100	1	
12	STORE_NAME		90	1	0 (sp),
15	LOAD_CONST		100	2	
18	BUILD_TUPLE		102	0	object (sp),
21	LOAD_CONST		100	3	object, () (sp),
24	MAKE_FUNCTION		132	0	object, (), CodeObject:stdlib_object (sp),
27	CALL_FUNCTION		131	0	object, (), PyFunction:object (sp),
Execution trace of object					
3	LOAD_GLOBAL		116	0	object, (), -1, PyFunction:object (bp) (sp),
6	STORE_NAME		90	1	object, (), -1, PyFunction:object (bp), object (sp),
7	LOAD_LOCALS		82		object, (), -1, PyFunction:object (bp) (sp),
8	RETURN_VALUE		83		object, (), -1, PyFunction:object (bp), CodeObject:stdlib_object (sp),
28	BUILD_CLASS		89		object, (), CodeObject:stdlib_object (sp),
31	STORE_NAME		90	2	PyClass:object (sp),
34	LOAD_CONST		100	4	

Figure 6: Screenshot of single test.

In addition to the html interface, we set up scripts for running the tests and benchmarks in the console using Google's V8 to run our VM, as well as directly in Python.

The setup was fairly elegant, so tests could be added just by adding the file containing the test code and running a single script (`tools/createTest.py` - see section B).

After having established this small test framework, our development was almost purely test-driven. This made development somewhat more relaxed as the tests provided confidence that a given modification actually worked. The tests also encouraged us to a more experimental approach to possible modifications - making a change and testing it was often much quicker than figuring out if it would work in every situation. Finally it was also quicker for us to correct errors, as the test overview and especially the single test trace made it easy to identify problems.

5 Current status

5.1 Supported language

In this section we describe what parts of the Python language our VM supports.

We decided on implementing Python 2.4.3, as that was the version installed on the department's machines.

Everything described here is assured to some degree with test cases, but it should be noted that there is lots of room for more thorough testing. This also means that our claims of supported features below only holds to the extent ensured by our tests. At the deadline, we had 69 test cases and 36 benchmarks. Examples of unsupported features can be seen in `test/unsupported.`

It should also be mentioned that we have no negative test cases and have not attempted to catch illegal programs in our implementation. Thus, running an illegal program will cause undefined behavior.

5.1.1 Basic arithmetics

Basic arithmetic operations were fairly easily implemented, as we just used the corresponding JavaScript operators. This also means that the semantics have not been ensured any further than what is specified in the test cases. Longs are not supported, but floats are. The implementation doesn't quite match Python's apparent intention of making objects out of everything, which would imply that even the basic arithmetic operators should be implemented as functions or methods.

5.1.2 Basic constructs

Conditionals and loops are supported fully, and especially loops have been tested reasonably well. We support both the `range` and the `xrange` constructs which are typically used in `for`-loops. The semantics are correct, as `range` actually produces a list while `xrange` is implemented in a more space efficient manner which generates each index lazily. Both are implemented in the library, purely in Python.

5.1.3 Functions/methods

We implemented functions before classes, in order to be ready to implement methods when introducing classes. Functions and methods are fully supported, including nesting and recursion. Keyword arguments and default arguments are working, and so is optional parameters. That is, both * and ** work and can be used. When defining functions, * creates lists out of actual parameters while ** creates dictionaries out of keyword parameters. When calling functions, * is used for unfolding a list into parameters, while ** unfolds a dictionary.

5.1.4 Classes and objects

Classes and objects are supported. We don't fully support the correct semantics when dynamically injecting fields on classes. In Python, update or addition of a field on a class is automatically pushed to all instances of the class, overwriting or marking dirty every instance's value for that field. We have not implemented this behavior, as that would require every class to hold references to all instances of it. This is of course possible, but we decided not to implement it partly because we discovered the issue shortly before deadline and partly because it would not be very elegant.

Inheritance was fairly easy to add on top of the basic class/object implementation, as Python's inheritance rules are reasonably simple. They simply state that the ordered list of base classes should be traversed and the first field/method with matching name should be chosen. This is working as it should which is ensured by a couple of tests.

It should be noted that we have by no means examined every corner of the language, and therefore there probably are subtle (and less subtle) object related features we don't handle.

5.1.5 Lists, tuples and dictionaries

These are the basic data structures in Python. They have all been implemented directly in JavaScript since Python has no arrays on which to build. For more information see section 4.6.4.

5.1.6 Iterators

In Python, any object may be *iterable,* meaning that it can return an iterator that can enumerate the object in some sensible way, using the methods `next()` and `getItem(...)`. We've implemented the iterator for lists and tuples which works as it should. It also works for general objects, where the object itself is responsible for implementing the actual iterator.

5.1.7 Exceptions

We have very basic support for exceptions. A special case handles the StopIteration exception which is necessary for iterators to work. Aside from that, using the `raise` keyword with no argument behaves correctly together with `try` and `except`. It should also be possible to give an argument to `raise`, which will simply be ignored, but this has not been tested.

5.2 Optimizations

5.2.1 Stack

In order to allow JavaScript to garbage collect, we implemented a new stack described in 4.6.1. The old one is described in the following.

We implemented the stack as a JavaScript object with an array containing the elements of the stack. The stack object also has a bottom pointer(BP) and a stack pointer(SP) pointing to the top and bottom of the current stackframe respectively. When we call and return from functions we call the removeFrame and newFrame methods, these modify the stack as shown in figure 7.

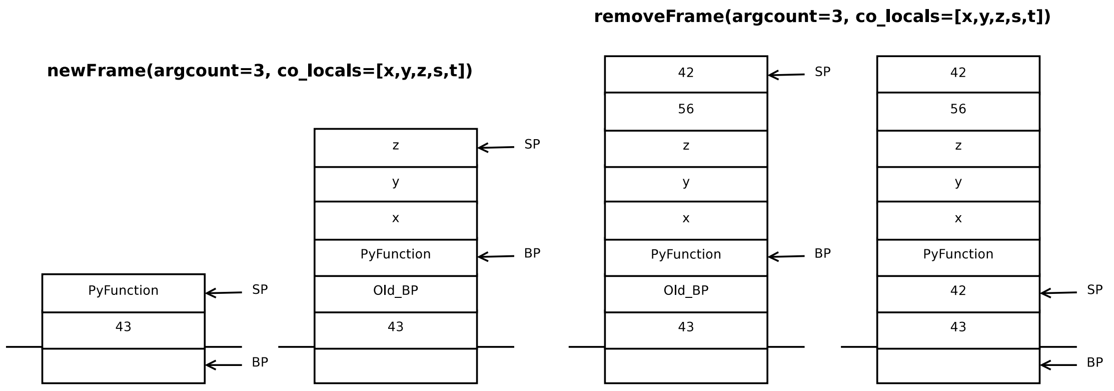


Figure 7: Stack example

On removeFrame we syncronize the localvariables with the arguments on the stack and sets TOS to the return value. From figure 7 it shows that we don't delete things from the stack again, we just overwrite the contents when pushing data onto the stack. It would have been advantageous to delete data over the SP when returning from a method. This would allow the JavaScript VM to garbage collect the used data.

5.2.2 Jumps

A jump bytecode has an argument telling where to jump to. In the first version of our code object the `co_code` array, which contains the opcodes and arguments, contained each instruction as a nested array. This meant that we could not jump directly to that index calculated by the compiler, since we had obscured the instruction array. Instead we needed to search through each instruction in the array to see if this had the offset we were looking for. This original approach had the advantage that we could have the instruction name and other information available, which helped us understand the programs, but in an attempt to make it faster, we changed it.

The approach was to create a single array with the opcodes and arguments. An example of the `co_code` array can be seen in algorithm 2. This allowed us to jump directly to the instruction given by the argument to the jump bytecode. We were expecting a big improvement in performance as jumping directly must be way faster than looking through an array. In practice we only saw a slight increase in performance based on the bencmarks in the test suite, from around 32 seconds to 31 seconds. We think that is partly due to the structure of the tests where we don't need to jump very far and therefore don't need to scan for the right offset for very long, and partly because we have introduced a check in each fetch-decode-execute cycle.

5.2.3 Refactoring

When we started the implementation, many of the elements from figure 3 was implemented as globals. This was obviously a poor choice and we decided to refactor and introduce the structure we have now. We expected to see a performance decrease since lookup of the elements would be more indirect. It turned out that the refactoring had a huge positive impact on performance. We went from times around 31 seconds to 11 seconds to execute our benchmarks. We have discovered that this is because globals take a very long time to resolve.

5.2.4 JIT compilation and peephole optimization

Shortly before the deadline, we tried, as an experiment, to implement a sort of JIT compilation. This works as follows.

In the interpreter switch, it is clearly possible just generating and saving JavaScript code as a string instead of actually executing it. This way, the Python bytecodes are parsed, resulting in an equivalent JavaScript program which is then evaluated. This should be somewhat faster because the program string only is generated once - subsequent calls to the same object are evaluated immediately. Generating code this way would also enable us to do peephole optimization on our generated code, using string parsing.

The obstructing complication is jumps. JavaScript doesn't have have labels and gotos, and therefore we are forced to bail out to interpret mode whenever a code object contains jumps.

An improvement would be a more finegrained compilation, where we just generate code between jumps and let jumps be handled in the interpreter. We have not had time for trying this.

5.3 Benchmarks

Here we provide performance measurements of execution times. We used V8 0.4.5 (candidate) and Firefox 3.0.4 with SpiderMonkey 1.8.1.13. The benchmarks are taken from the pybench benchmark suite.

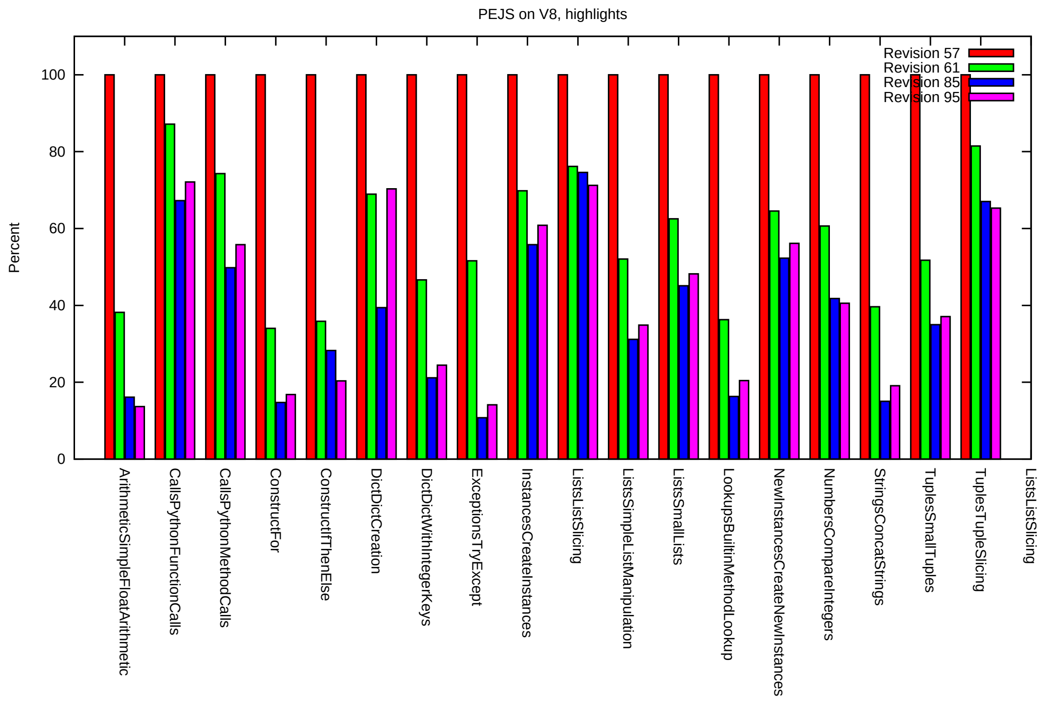


Figure 8: Selected benchmarks on PEJS revisions

In figure 8. Revision 57 is the original unoptimized version, revision 61 is refactoring, see 5.2.3. Revision 85 is refactoring of a debug variable to be local instead of global. revision 95 is after the new stack was implemented, see 5.2.1. For simplicity we have included a graph with selected benchmarks, the full benchmark is in appendix C. We note that the new stack doesn't improve performance in all cases. The cases with many method calls are penalized by having to create a new stack frame for every method invocation.

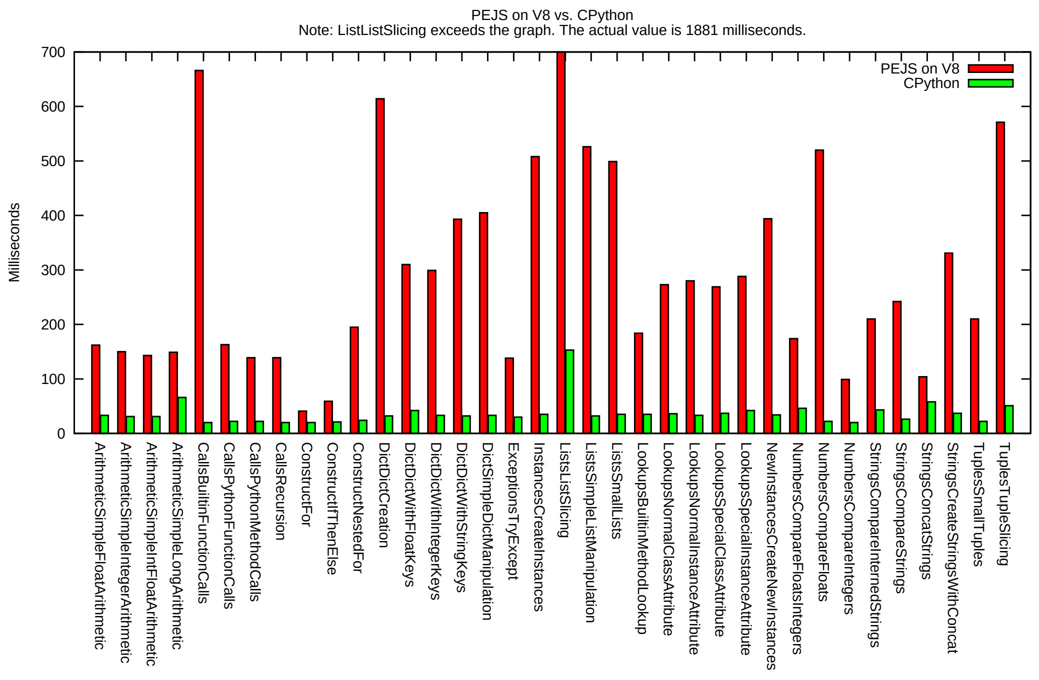


Figure 9: PEJS compared to CPython

In figure 9 we benchmark PEJS against CPython, it shows that we are approximately a factor 2 to 20 slower.

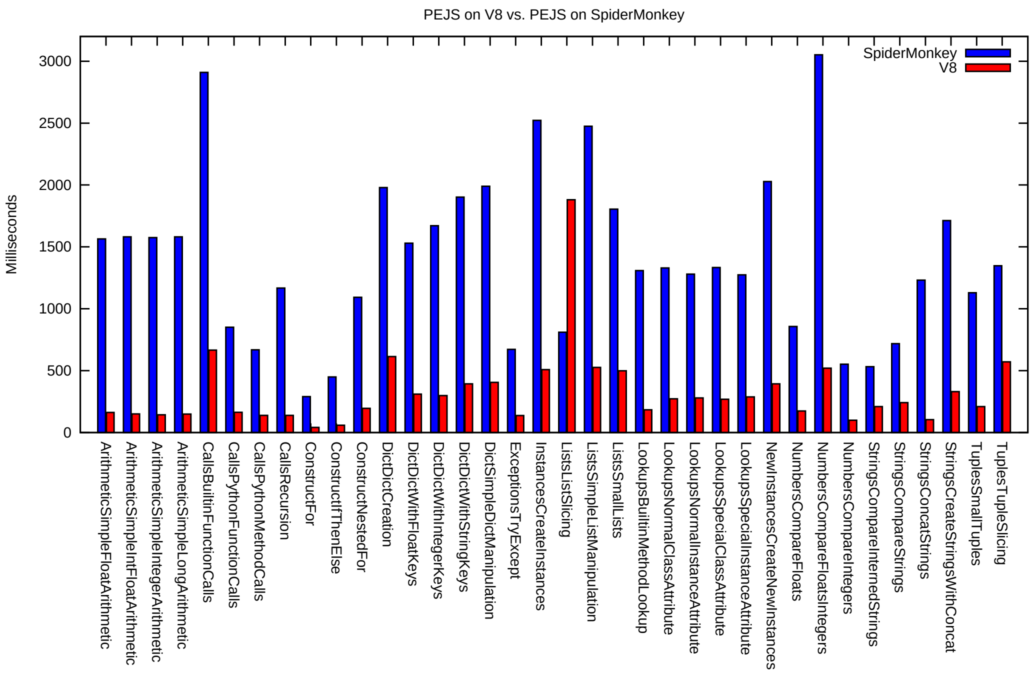


Figure 10: V8 vs. SpiderMonkey

In figure 10 we run PEJS on SpiderMonkey an V8, it shows that V8 is much faster in the general case, except in ListListSlicing where SpiderMonkey is twice as fast. This is probably due to a slow implementation of `Array.slice(...)` and `Array.splice(...)` in V8.

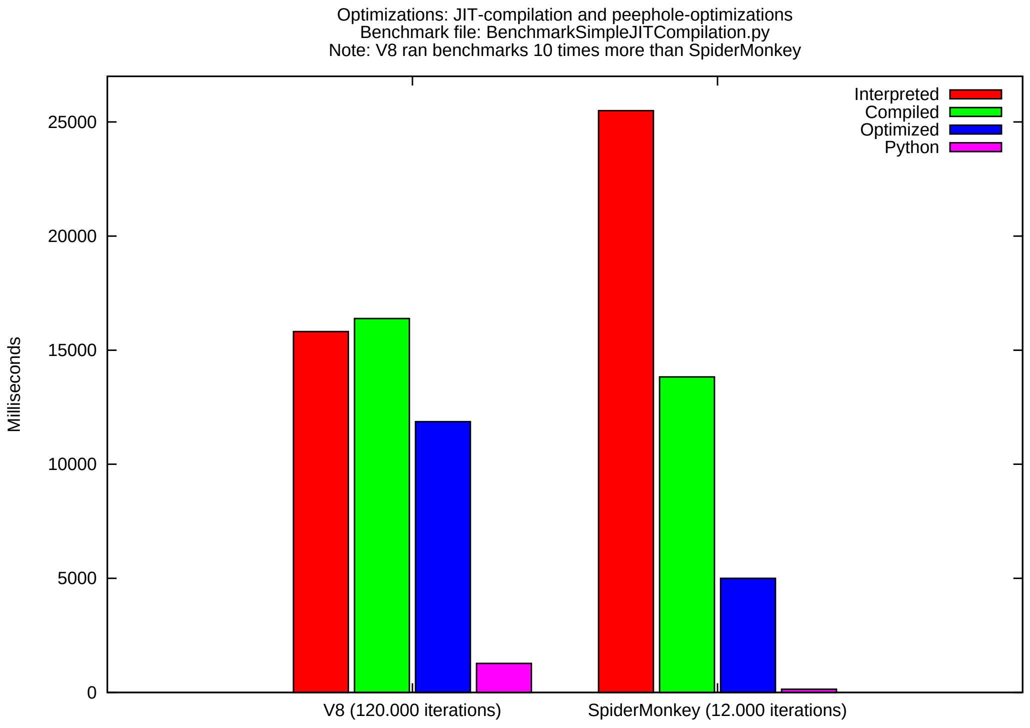


Figure 11: JIT compilation benchmark

In figure 11, we present the benchmarks for JIT compiling a specially generated benchmark. First column is without JIT. Second coloumn is with JIT. Third coloumn is with JIT and peephole optimization. Note that the peephole patterns is designed specifically for this case. SpiderMonkey earns by the JIT compilation while it seems that V8 is doing something similarly clever already. Of course, the optimizations help both.

5.4 Alignment with goals

Here we reflect on our progress in comparison with our initial plan (3) and in relation to the course objectives.

When planning implementation in JavaScript, it wasn't completely clear to us how we would or should handle heap representation and garbage collection. This turned out to be completely trivial as we chose to simply implement Python objects and code objects as regular JavaScript objects. The consequence of this was that no heap and no garbage collector was necessary, nor would they make any sense. This has definitely been a great help in having time for implementing a larger subset of our target language, but on other hand we have not gained much experience with regard to heap design and garbage collection.

We were pleasently surprised of how easy it was to establish a library. When starting development, we were not at all sure how or if we would be able to get a library running. This turned out not to be a problem, but we have not had time for extending it very much.

The only thing we're not satisfied with is the fact that the performance of our VM is significantly lower than that of CPython. We would have liked to optimize our code some more, but we were not able to find a decent profiler for JavaScript. FireBug for Firefox provided information in a useful format, but the results were unreliable. V8's profiler wasn't really useful as it provided information mostly on the level of opcodes, which wasn't high level enough to extract sensible information on where time was spent in our code. Thus we could not identify time consuming code any better than by examining the code manually.

5.4.1 Course objectives

The only aspect of our project that relates directly to any of the course objectives is the fact that we have implemented a variant of a JIT compiler. Other than that we have gained a lot of experience on how to implement a VM, allthough the project doesn't match the very specific course objectives precisely.

6 Future work

In this section we outline possibilities for future work on our VM.

6.1 Optimizations

6.1.1 Switch-case as array

One attempt to minimize the interpreter overhead could be to replace the switch-case with an array containing JavaScript functions corresponding to the code executed in each case. The array indexes should match the case number, so a fetch-decode-execute cycle would look like sketched in algorithm 3, where `functions` is the array replacing the switch-case.

Algorithm 3 Outline of the switch-case as array approach

```
while(true) {
  bytecode = prog[pc];
  if (bytecode >= 90) {
    argument = prog[pc+2];
    pc += 3;
  } else {
    pc++;
  }
  functions[bytecode](argument);
}
```

This approach will introduce an extra function call per fetch-decode-execute cycle, but assuming constant-time lookup in the array, it might be feasible. The actual impact should be benchmarked.

The approach has many resemblances to the threaded code approach, as described by Bell[3], but with larger overhead.

6.1.2 Optimizations on byte code order and format

Throughout the project, we have relied on the order in which the Python compiler outputs byte codes. In some cases, the way we handle byte codes does not correspond to the optimal order, in which elements could be placed on the stack. In the example shown as algorithm 4, three elements are placed on stack in the order shown in figure 12. The present stack order makes it necessary to use two named local variables, instead of supplying the arguments unnamed to the JavaScript `slice` method as shown.

Algorithm 4 Examples of present and optimized byte code order usage.

```
//Present byte code order:
case 33: //SLICE+3
  var end = stack.pop();
  var start = stack.pop();
  stack.push(new this.types.PyList(stack.pop().store.slice(start,end)));
  break;

//Optimized byte code order:
case 33: //SLICE+3
  stack.push(new this.types.PyList(stack.pop().store.slice(stack.pop(),
                                                          stack.pop())));
  break;
```

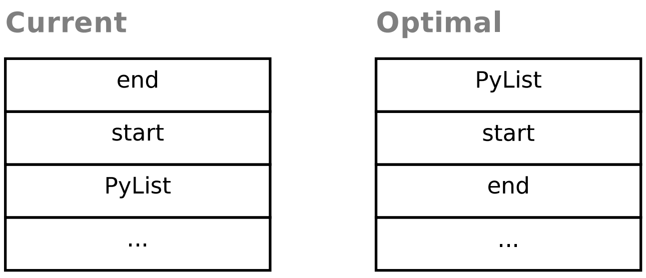


Figure 12: Stack in present an optimized state before a SLICE+3 byte code

Becuase JavaScript is a high-level language, the impact of this suggested optimization is hard to predict, as a JavaScript compiler could optimize the present implementation to be as performant as the suggested, optimized implementation.

6.1.3 Library snapshot

Upon startup of an execution we interpret the standard library every time. Since the library is pretty static it could be advantageous to create a snapshot of the library that could just be loaded instead of interpreting the library declarations every time. This would decrease load time,

[3]Threaded Code, James R. Bell, Communications of the ACM, June 1973

especially as the library's size increases. In the current state the library is fairly small so the penalty is not that big.

6.2 Python compiler

Implementing a Python compiler in JavaScript that outputs the code objects we need would allow us to have Python embedded in a html document. There is no technical obsacles in the way of doing this, and if PEJS is to have much use in the real world this would definitely be needed. We have focused on the VM since that is what this course is about, but it could be a nice sparetime project.

6.3 Library extension

The library could be extended to implement a larger part of the standard Python library. More interestingly, it would be relevant to implement a library for interacting with the DOM in the browser. This would make it very feasible to use Python for writing web pages. It would have to be done mostly in JavaScript, in order to utilize JavaScript's DOM functionality.

7 Conclusions

We can see that the our goal of being able to embed Python directly in html and use it for implementing web pages is actually realistic. We only need a Python compiler written in JavaScript and a Python library for interacting with the browser DOM. It is definitely possible to build a Python compiler in JavaScript, and implementing a DOM library is also feasible. It can simply be written as wrapper code on the JavaScript functions.

This project has shown that it actually is possible to make a simple VM and it's not that difficult. This was a bit surprising for us, as we didn't expect the development to go so smoothly as it has. Also, building a VM on top of JavaScript is beginning to be feasible as Javascript VM's improve. The current V8 actually runs the code in a relatively acceptable speed, depending on demands. The benchmarks showing differences between V8 and SpiderMonkey clearly illustrates that the speed of the underlying VM has a big impact on PEJS performance, so as JavaScript VM's improve so will PEJS.

A Directory structure

For convenience we provide an overview of the directory structure and content and purpose of the various files in the project. Note that unimportant folders and files have been omitted. The project files are available at `http://code.google.com/p/pejs`.

Algorithm 5 Directory and file structure.

```
src/
  lib/                    Library functions, implemented in Python.
    stdlib.py             Python standard library, imported by default.
    string.py             Python string library.
    time.py               Python time library.
  interpreter.js          All of the actual virtual machine.
test/                     Tests and benchmarks. Note that benchmarks look like
                          tests, but don't actually test anything.
  unsupported/            Tests and benchmarks not currently supported.
    <test or benchmark>.py
  test.html               Open this to run tests and see results in a browser
                          (requires tools/createTest.py to be run first).
  testBenchmark.html      Run benchmarks in browser, output formatted for
                          copying to spiderlog.txt (accessible from test.html).
  testSingle.html         Run single test and show execution and stack trace
                          (should be accessed from test.html).
  <test or benchmark>.py  Actual tests and benchmarks.
tools/                    Utility scripts. Should be invoked from root folder
                          (e.g. ./tools/createTest.py).
  prepareStandalone.js
  prepareStandaloneBenchmark.js
  compiler.py             Uses the Python compiler to translate from Python
                          source code to JavaScript containing Python bytecodes.
  createTest.py           Translates all tests and benchmarks.
                          Necessary for test.html to work.
  benchmarkInV8.sh        Performs benchmark in V8 and appends the result
                          to v8log.txt.
  cleanup.sh              Removes all generated files.
  prepareStandalone.sh
  prepareStandaloneBenchmark.sh
  testInPython.sh         Runs tests in Python, outputting results to console.
  testInV8.sh             Runs tests in V8, outputting results to console.
spiderlog.txt             Benchmark results from Firefox.
v8log.txt                 Benchmark results from V8.
```

B How to run

Note: Our VM has been tested on SpiderMonkey and V8 (both standalone and in Firefox/Chrome). It doesn't run in Internet Explorer.

Prerequisites

- The py-files should be run with Python 2.4.3 installed. In the zip-file provided, it is not necessary to run the createTest-script, we have already translated the tests.
- The bash scripts should be run in a Linux console.

Run tests

In order to see the tests/benchmarks run:

1. execute `tools/createTest.py`
2. view `test/test.html` (click each test to see execution trace)

Add a test

1. Create .py file in test/ (make sure it prints 42 at the end)
2. execute `tools/createTest.py`
3. view `test/test.html`

Embed in html

1. Create and write `"myProgram.py"` file
2. Translate with `"tools/compiler.py myProgram.py"`.
3. Put lines

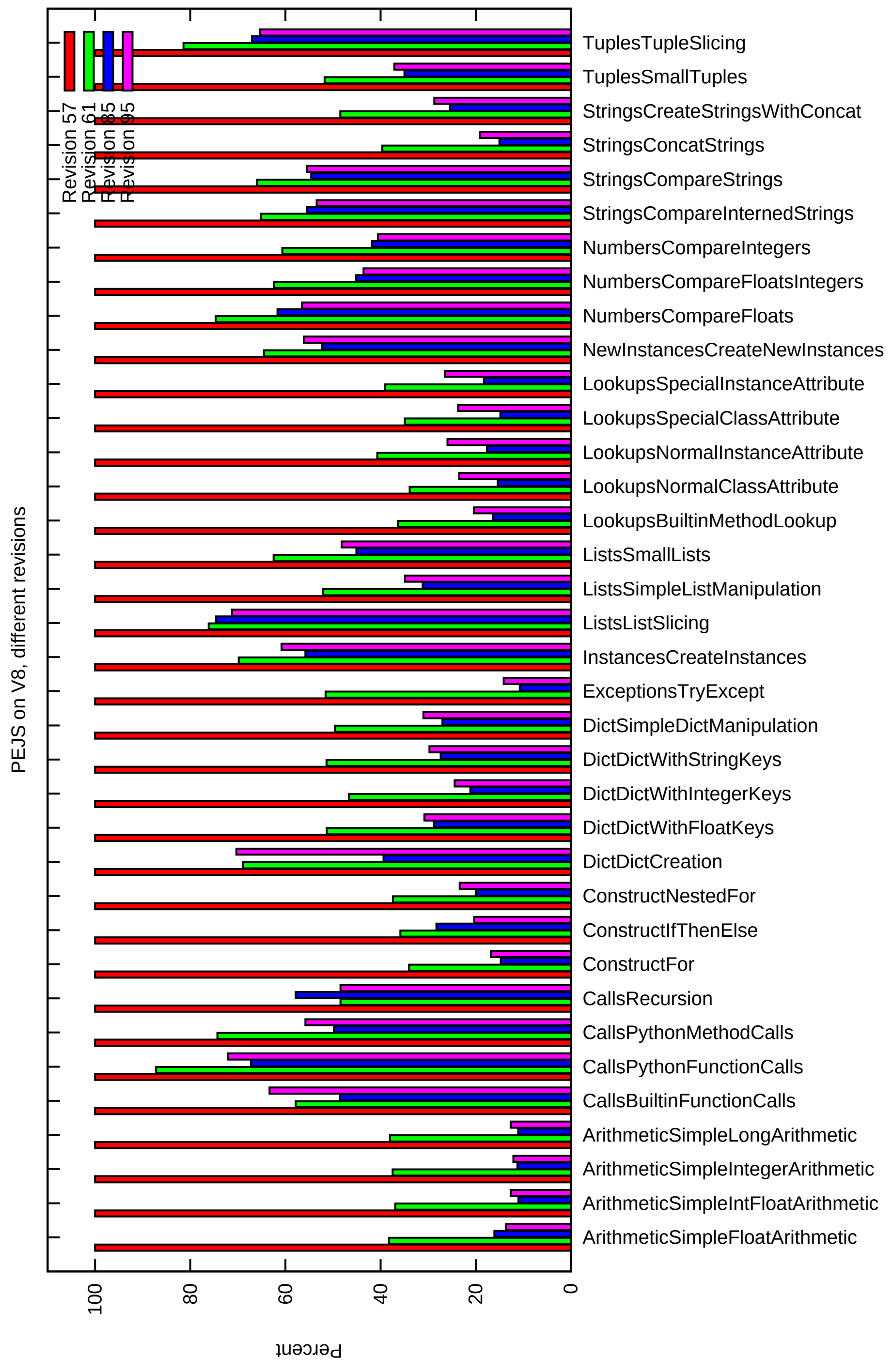
   ```
   <script type="text/javascript" src="pejs/src/interpreter.js"></script>
   <script type="text/javascript" src="pejs/src/lib/stdlib.js"></script>
   <script type="text/javascript" src="myProgram.js"></script>
   ```

   in the head of an html file.
4. Put

   ```
   <script type="text/javascript">
   (new PEJS()).interpret("myProgram");
   </script>
   ```

   in body of html.

C Additional benchmark


PEJS on V8, different revisions
Revision 57
Revision 61
Revision 85
Revision 95
TuplesTupleSlicing
TuplesSmallTuples
StringsCreateStringsWithConcat
StringsConcatStrings
StringsCompareStrings
StringsCompareInternedStrings
NumbersCompareIntegers
NumbersCompareFloatsIntegers
NumbersCompareFloats
NewInstancesCreateNewInstances
LookupsSpecialInstanceAttribute
LookupsSpecialClassAttribute
LookupsNormalInstanceAttribute
LookupsNormalClassAttribute
LookupsBuiltinMethodLookup
ListsSmallLists
ListsSimpleListManipulation
ListsListSlicing
InstancesCreateInstances
ExceptionsTryExcept
DictSimpleDictManipulation
DictDictWithStringKeys
DictDictWithIntegerKeys
DictDictWithFloatKeys
DictDictCreation
ConstructNestedFor
ConstructIfThenElse
ConstructFor
CallsRecursion
CallsPythonMethodCalls
CallsPythonFunctionCalls
CallsBuiltinFunctionCalls
ArithmeticSimpleLongArithmetic
ArithmeticSimpleIntegerArithmetic
ArithmeticSimpleIntFloatArithmetic
ArithmeticSimpleFloatArithmetic
Percent