JSish

Ryan Grasell

June 2015

1 Introduction

For my senior project, I implemented Professor Keen's JSish spec in C++. JSish is a subset of Javascript with support for execution from the command line and files.

1.1 Educational Goals

I chose this project because I wanted to reinforce my programing language skills, learn C++, and gain a deeper understanding of garbage collection.

I quickly grew familiar with C++. My prior experience with both C and Java was a good enough introduction to let me hit the ground running. The biggest hurdle for me was the lack of garbage collection built into C++. I understood the principles of manual memory management, but I needed to grow accustomed to manually freeing memory in a complex, object oriented environment.

The most educational part of this project was writing the garbage collector. It was both an illuminating experience because I had never written one before, and very practical because it gave me insight to how modern languages run.

1.2 Testing

I set up a testing environment with 2 tiers: unit and integration tests.

Unit testing covers each class individually. Each expression, statement, value, and feature are covered independently. Unit tests are run with the Catch C/C++ testing library.

Integration tests cover the behaviour of the project as a whole. The integration tests run the interpreter against sample JSish files, and compare the outputs to known correct output.

The combination of unit and integration tests gives very good test coverage to the project.

2 Performance Analysis

One of the draws of C++ is the potential for speed. Using C++ instead of a more specialized language like SML adds complexity to development, and should only be used if there are benefits.

I profiled 2 use cases for JSish. Small, responsive programs and longer running computational programs. To benchmark these use cases, I measured startup time for the interpreter, and duration of the program respectively.

Each benchmark was run by an SML implementation of JSish, and various builds of the C++ implementation (each compiled with a different gcc optimization level).

2.1 Startup Time

I tested startup time by running each interpreter against a test file whose only statement was

```
print ''Hello, World!'';
```

This test was designed to measure only how long it took the interpreter to begin executing code.

The results of this benchmark do not lend themselves well to graphical representation. The SML implementation took **1070ms** to start up, while all variations of the C++ implementation took only **2ms**.

The SML implementation is not suitable for small command line programs. However, the C++ implementation starts up fast enough to not interrupt a user's workflow.

2.2 Computational Performance

The other class of benchmarks I ran cover long-running computational problems. The reported numbers are the combined running times of all benchmarks.

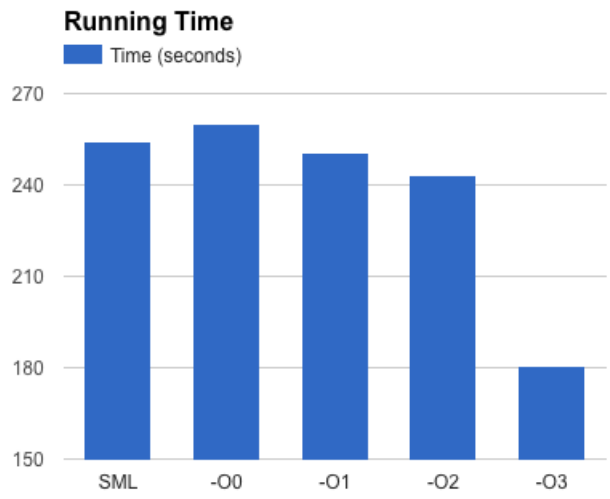

Figure 1: Running time for benchmark suite

GCC (with optimization) produced very good results. Especially with -O3 optimizations, the C++ implementation of JSish overtakes SML significantly.

2.3 Conclusion

Both benchmarks show that the C++ implementation is significantly faster than SML. The extended development time for the C++ implementation is worthwhile, as long as one has access to a good optimizing C++ compiler.

3 Optimization

The main draw of C++ as the implementation language of JSish was speed. The architecture of my C++ JSish implementation requires a high degree of object allocations and deallocations.

I measured memory allocation performance by running the computational benchmarks and recording function call durations with Callgrind.

A large percent of time in my computational benchmark was spent in memory allocation and deallocaiton.

To mitigate the heavy cost of memory allocation, I implemented a free list for all of the garbage-collectable classes in the project. This significantly cut down on memory management time:

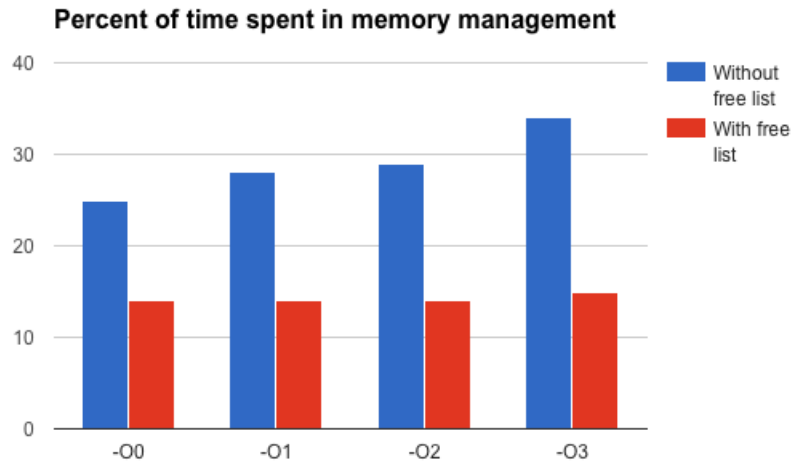

Figure 2: Percent of time spent in memory management

GCC's optimizations were able to cut down the running time for the overall program, but they did not seem to speed up memory management. This made it even more critical to implement a good optimization.

The free list did significantly speed up runtime for JSish. This is definitely a case where C++'s power gave good results.

4 Reflection

Over the course of this project, I discovered the pros and cons of the different methods and tools I used. At the beginning of the project, I made choices based on their educational value more than their strict effectiveness to my project. If I were to redo this project, there are a few decisions I would make differently:

4.1 Language

C++ was a good choice for this project. First of all, it was a great learning experience for me. My previous exposure to C++ was very small, and I picked up a good working knowledge of the language. C++ was well suited to the problem; it was very performant as discussed in the 'performance' section. However, it also posed a few challenges:

4.1.1 STL

C++'s Standard Template Library formed the backbone of my data structures. I found it powerful and fast, but cumbersome to work with. STL containers were often verbose. For example, checking for membership in a set required:

Other tests for set membership, like the [] operator, would automatically create, insert, and return a new member for the set if it didn't already exist. To STL's credit, that behaviour was well documented. Personally, I found in counterintuitive.

```
for (GarbageCollectable *g : allObjects) {
    if (seen.find(g) == seen.end()) {
        toErase.insert(g);
    }
}
```

Figure 3: Testing for Set Membership

4.1.2 Portability

I specifically targeted Linux and Mac OS for this project. Even though I specifically used only POSIX and cross-platform libraries, I experienced some trouble with compatibility.

The GNU Readline library is supposed to be supported on both Mac OS and Linux. I found that to be mostly true, despite a few graphical glitches. The workarounds were straightforwards, but had I chosen to also target Windows, using Readline would have been much more difficult.

Overall, C++ gave me decent portability inside the POSIX family of operating systems. Porting to Windows would have been an ordeal; either I would have to convert my system calls to Window's, or the user would have to install a POSIX emulation layer.

This problem could be mitigated with better software design in my project. However, time constraints forced me to focus only on POSIX compatibility. Using a managed language may have given me that cross-platform compatibility for no extra development time.

4.1.3 Conclusion

C++ performed well for this project. However, a managed language could have solved the problems I ran into. My preference for Java may just be personal bias, but I would have been much more productive using Java's Collections API. Java's portability would definitely have simplified the process of porting

my project to other platforms.

C++'s saving grace is the raw speed and the flexibility it offered to the developer. C++ made it simply to implement the free list without having to do major code architecture changes.

I would have liked to implement JSish in Java to explore the performance and practicality of the language, but again I hit a time constraint. I believe Java could offer solutions without many drawbacks, especially with its built-in garbage collection.

4.2 Code Architecture

This project has many moving parts, and good execution required modular design. I met the modular design requirements by breaking the code up into several packages:

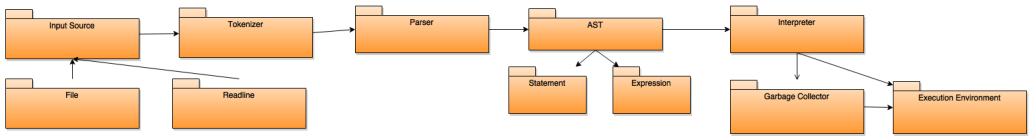

Figure 4: Project Architecture

This design worked well during development. Near the end of development, I was able to add Readline support easily. Other additions, like a garbage collection statement, were also simple to add and required minimal code changed. Adding the 'gc' statement took only 10 lines of code.

The design I settled on has at least one major flaw: code for execution and garbage collection for each statement and expression is kept in the class files.

This made development simple at first, but as I added features the files became unwieldy. Files got too large to easily read through, and it was hard to make changes to interfaces. Because code was so spread out, changes had to

be made in dozens of files. The observer pattern would have kept kept each file to a minimal size and localized changes. On a development team of more than one person, the observer pattern would also have made collaboration easier. Refactoring this code to the observer pattern would be a simple, but possibly time consuming endeavor.

4.3 Build System

I used a makefile to build JSish. It definitely had its benefits: I was able to use the same build system on both Mac OS and Linux and it performed both incremental and multithreaded builds. The makefile also builds and runs the test cases.

However, it was difficult to put together (although that was a one time effort) and it often had to rebuild the project from scratch when large changes were made.

Makefiles also have portability issues. My simple makefile expects libraries and binaries to be in standard locations. Any more complicated configurations would require Autotools or a similar system. Autotools is overly complicated for a project of this scope, but there do not seem to be any better systems to work with makefiles.

4.3.1 Alternative Build Systems

If I were to start this project again, I would choose to use a dependency-managing "next generation" build system.

Maven and Gradle have automatic dependency resolution that would greatly simplify configuration of the build environment. For example, a Maven build could specify the testing library by simply adding an entry to the configuration XML file. Using the makefile build, I had to add all of the testing library code

to the project and manually point to it in the makefile.

Maven has a native build plugin called NAR. NAR supports building native code, handling environment specific details, and running test cases. If I were to continue development on this project, my first step would be to transition the build to Maven/NAR.

5 Conclusion

This was my first attempt at a C++ project, and I dove deep enough into the language to write the parser and interpreter. Implementing the JSish in a general purpose language was good practice, and the skills could easily be useful in the future. Implementing my own garbage collector gave me a good overview of garbage collector internals. In the future, most code I write will run in a garbage-collected environment so this experience will help me create better performing products.

JSish was an excellent educational experience. I set out to reinforce my programing language skills, learn C++, and gain a deeper understanding of garbage collection. I consider this project a success on all three counts.