JSZap: Compressing JavaScript Code

Martin Burtscher
University of Texas at Austin
burtscher@ices.utexas.edu

Benjamin Livshits
Microsoft Research
livshits@microsoft.com

Gaurav Sinha
IIT Kanpur
gsinha@iitk.ac.in

Benjamin G. Zorn
Microsoft Research
zorn@microsoft.com

Microsoft Research Technical Report
MSR-TR-2010-21

Abstract

JavaScript is widely used in web-based applications, and gigabytes of JavaScript code are transmitted over the Internet every day. Current efforts to compress JavaScript to reduce network delays and server bandwidth requirements rely on syntactic changes to the source code and content encoding using gzip. In this paper, we consider reducing the JavaScript source code to a compressed abstract syntax tree (AST) and transmitting the code in this format. An AST-based representation has a number of benefits including reducing parsing time on the client, fast checking for well-formedness, and, as we show, compression.

With JSZAP, we transform the JavaScript source into three streams: AST production rules, identifiers, and literals, each of which is compressed independently. While previous work has compressed Java programs using ASTs for network transmission, no prior work has applied and evaluated these techniques for JavaScript source code, despite the fact that it is by far the most commonly transmitted program representation. We show that in JavaScript the literals and identifiers constitute the majority of the total file size and we describe techniques that compress each stream effectively. On average, compared to gzip we reduce the production, identifier, and literal streams by 30%, 12%, and 4%, respectively. Overall, we reduce total file size by 10% compared to gzip while, at the same time, benefiting the client by moving some of the necessary processing to the server.

1 Introduction

Over the last decade, JavaScript has become the lingua franca of the web, meaning that increasingly large JavaScript applications are being delivered to the user over the wire. The JavaScript code of large applications such as Gmail, Office Web Apps, and Facebook amounts to several megabytes of uncompressed and hundreds of kilobytes of compressed data.

This paper argues that the current Internet infrastructure, which transmits and treats executable JavaScript as files, is ill-suited for building the increasingly large and complex web applications of the future. Instead of using a flat, file-based format for JavaScript transmission, we advocate the use of a hierarchical, abstract syntax tree-based representation.

As prior research by Franz et al. [4, 25] has argued that switching to an AST-based format has a number of valuable benefits, including the ability to quickly check that the code is well-formed and has not been tampered with, the ability to introduce better caching mechanisms in the browser, etc. However, if not engineered properly, an AST-based representation can be quite heavy-weight, leading to a reduction in application responsiveness.

This paper presents JSZAP, a tool that generates and compresses an AST-based representation of JavaScript source code, and shows that JSZAP outperforms de facto compression techniques such as gzip [16] by 10% on average. The power of JSZAP comes from the fact that JavaScript code conforms to a well-defined grammar [18]. This allows us to represent the grammar productions separately from the identifiers and the literals found in the source code so that we can apply different compression techniques to each component. With JSZAP, we transform the JavaScript source into three streams: AST production rules, identifiers, and literals, each of which is compressed independently. While previous work has considered compressing Java programs for network transmission [4, 24, 25, 43], no prior work has considered applying these techniques to JavaScript. We show that in JavaScript the literals and identifiers constitute the majority of the total file size and describe techniques that compress each stream effectively.

1.1 Contributions

This paper makes the following contributions.

- It demonstrates the benefits of AST-based JavaScript program representations, ranging from the ability to parse code in parallel [28] to entirely removing blocking HTML parser operations and better caching, leading to more responsive web applications.
- It introduces JSZAP, the first grammar-based compression algorithm for JavaScript. JSZAP represents productions, identifiers, and literals as independent streams and uses customized compression strategies for each of them.
- It evaluates JSZAP on a range of nine JavaScript programs, covering various program sizes and application domains, and ranging in size between about 1,000 to 22,000 lines of code. We conclude that JSZAP is able to compress JavaScript code 10% better compared to gzip. JSZAP compression ratios appear to apply across a wide range of JavaScript inputs.

1.2 Paper Organization

The rest of this paper is organized as follows. Section 2 provides background on JavaScript and AST-based program representation. Section 3 gives an overview of AST compression and Section 4 goes into the technical details of our JSZAP implementation. Section 5 presents the evaluation methodology and our experimental results. Section 6 describes related work and Section 7 concludes.

2 Background

This section covers the fundamentals of how JavaScript-based web applications are constructed and then advocates an AST representation as a transfer format for JavaScript.

2.1 Web Application Background

Over the last several years, we have witnessed the creation of a new generation of sophisticated distributed Web 2.0 applications as diverse as Gmail, Bing Maps, Redfin, MySpace, and Netflix. A key enabler for these applications is their use of client-side code—usually JavaScript executed within the web browser—to provide a smooth and highly responsive user experience while the rendered web page is dynamically updated in response to user actions and client-server interactions. As the sophistication and feature sets of these web applications grow, downloading their client-side code is increasingly becoming a bottleneck in both initial startup time and subsequent application reaction time. Given the importance of performance and instant gratification in the adoption of applications, a key challenge thus lies in maintaining and improving application responsiveness despite increased code size.

Indeed, for many of today's popular Web 2.0 applications, client-side components already approach or exceed one megabyte of (uncompressed) code. Clearly, having the user wait until the *entire* code base has been transferred to the client before execution can commence does not result in the most responsive user experience, especially on slower connections. For example, over a typical 802.11b wireless connection, the simple act of opening an email in a Hotmail inbox can take 24 seconds on the first visit. The second visit can still take 11 seconds—even after much of the static resources and code have been cached. Users on dial-up, cell phone, or other slow networks see much worse latencies, of course, and large applications become virtually unusable. Bing Maps, for instance, takes over 3 minutes to download on a second (cached) visit over a 56k modem. (According to a recent Pew research poll, 23% of people who use the Internet at home rely on dial-up connections [35].) In addition to increased application responsiveness, reducing the amount of code needed for applications to run has the benefit of reducing the overall download size, which is important in mobile and some international contexts, where network connectivity is often paid per byte instead of a flat rate.

From the technical standpoint, a key distinguishing characteristic of Web 2.0 applications is the fact that code executes both on the client, within the web browser, and on the server, whose capacity ranges from a standalone machine to a full-fledged data center. Simply put, today's Web 2.0 applications are effectively sophisticated distributed systems, with the client portion typically written in JavaScript running within the browser. Client-side execution leads to faster, more responsive client-side experience, which makes Web 2.0 sites shine compared to their Web 1.0 counterparts.

In traditional web applications, execution occurs *entirely* on the server so that every client-side update within the browser triggers a round-trip message to the server, followed by a refresh of the entire browser window. In contrast, Web 2.0 applications make requests to fetch only the data that are necessary and are able to repaint individual portions of the screen. For instance, a mapping application such as Google Maps or Bing Maps may only fetch map tiles around a particular point of interest such as a street address or a landmark. Once additional bandwidth becomes available, such an application may use speculative data prefetch; it could push additional map tiles for the surrounding regions of the map. This is beneficial because, if the user chooses to move the map around, surrounding tiles will already be available on the client side in the browser cache.

However, there is an even more basic bottleneck associated with today's sophisticated Web 2.0 applications: they contain a great deal of code. For large applications such as Bing Maps, downloading as much as one megabyte of JavaScript code on the first visit to the front page is not uncommon [30]. This number is for the initial application download; often even more code may be needed as the user continues to interact with the application. The vast amount of code is what motivates our interest in JSZAP.

2.2 Benefits of an AST-based Representation

Franz's Slim Binaries project was the first to propose transmitting mobile code in the form of an abstract syntax tree [25]. In that project, Oberon source programs were converted to ASTs and compressed with a variant of LZW [45] compression. In later work, Franz also investigated the use of ASTs for compressing and transmitting Java programs [4, 24, 43].

Since this original work, JavaScript has become the de facto standard for transmission of mobile code on the web. Surprisingly, however, no one has investigated applying Franz's techniques to JavaScript programs. Below we list the benefits of AST-based representation, both those proposed earlier as well as unique opportunities only present in the context of web browsers.

Well-formedness and security. By requiring that JavaScript be transferred in the form of an AST, the browser can easily and quickly enforce important code properties. For instance, it can ensure that the code will parse or that it belongs to a smaller, safer JavaScript subset such as ADSafe [13]. Furthermore, simple code signing techniques may be used to ensure that the code is not being tampered with, which is common according to a recent study [39].

Caching and incremental updates. It is common for large Internet sites to go through a large number of small code revisions. This kind of JavaScript code churn results in cache misses, followed by code retransmission: the browser queries the server to see if there are any changes to a particular JavaScript file, and if so, requests a new version of it. Instead of redelivering entire JavaScript files, however, an AST-based approach allows much more fine-grained updates to individual functions, modules, etc.

Unblocking the HTML parser. The HTML parser has to parse and execute JavaScript code synchronously, because JavaScript execution may, for instance, inject additional HTML markup into the existing page. This is why many pages place JavaScript towards the end so that it can run once the rest of the page has been rendered. An AST-based representation can explicitly represent whether the code contains code execution or just code declaration, as is the case for most JavaScript libraries. Based on this information, the browser should be able to unblock the HTML parser, effectively removing a major bubble in the HTML parser pipeline.

Compression. This paper shows that an AST-based representation can be used to achieve better compression for JavaScript, reducing the amount of data that needs to be transferred across the network and shortening the processing time required by the browser to parse the code.

2.3 JavaScript Compression: State of the Art

Currently, the most commonly used approach to JavaScript compression is to "minify" the source code by removing superfluous whitespace. JSCrunch [23] and JSMin [14] are some of the more commonly used tools for this task. Some of the more advanced minifiers attempt to also rename variables to use shorter identifiers for temporaries, etc. In general, such renaming is difficult to perform soundly, as the prevalence of dynamic constructs like `eval` makes the safety of such renaming difficult to guarantee. When considered in isolation, however, minification generally does not produce very high compression ratios.

After minification, the code is usually compressed with gzip, an industry-standard compression utility that works well on source code and can eliminate much of the redundancy present in JavaScript programs. Needless to say, gzip is not aware of the JavaScript program structure and treats it as it would any other text file. On the client machine, the browser proceeds to decompress the code, parse it, and execute it.

3 Compression Overview

This section gives an overview of the JSZAP approach, with Section 4 focusing on the details.

3.1 JavaScript Compression

JavaScript code, like the source code of most other high-level programming languages, is expressed as a sequence of characters that has to follow a specific structure to represent a valid program. This character sequence can be broken down into subsequences called tokens, which consist of keywords, predefined symbols, whitespace, user-provided constants, and user-provided names. Keywords include strings like `while` and `if`. Symbols are operators such as $-$ and $++$ as well as semicolons, parentheses, etc. Whitespace typically includes all non-printable characters but most commonly refers to one or more space (blank) or tab characters. User-provided constants include hardcoded string, integer, and floating-point values. User-provided identifiers are variable names, function names, and so on.

The order in which these tokens are allowed to appear is defined by the JavaScript grammar [18], which specifies the syntax rules. For instance, one such rule is that the keyword `while` must be followed by an opening parenthesis that is optionally preceded by whitespace. These syntax rules force legal programs to conform to a strict structure. In other words, randomly generated text files hardly ever represent correct JavaScript programs. A direct consequence of this structure is that JavaScript code is compressible. For example, the whitespace and the opening parenthesis after the keyword `while` are redundant. They convey no information because the whitespace is optional and the opening parenthesis must always be present. They are simply there to make the code look more appealing to the programmer. They can safely be omitted in a compressed version of the source code because the uncompressed source code can easily be regenerated from the compressed form by inserting an opening parenthesis after every occurrence of the word `while` (outside of string constants).

Because the compressed code is not directly executable but must first be decompressed, crunching tools like JSCrunch [23] and JSMin [14] do not go this far. They primarily focus on minimizing whitespace, shortening local variable names, and removing comments. As the `while` example above illustrates, whitespace is often optional and can be removed. In the cases where it cannot, it can at least be reduced to a single character. Comments can always be removed. Local variables can be arbitrarily renamed without changing the meaning of the program as long as they remain unique and do not collide with a reserved word or global name that needs to be visible. Crunching tools exploit this fact and rename local variables to the shortest possible variable names such as `a`, `b`, `c`, etc. The resulting code is compressed because it is void of comments and unnecessary whitespace such as indentation and uses short but meaningless variable names, making it hard to read for humans but not for machines. In fact, computers can read the crunched version more quickly because fewer characters need to be processed.

If we are willing to forego direct execution, i.e., to introduce a decompression step, we can achieve much higher compression ratios than crunching tools are capable of achieving. For example, general-purpose compressors such as gzip are often able to further compress crunched JavaScript programs by a large amount. In the case of gzip, recurring character sequences are compressed by replacing later occurrences with a reference to an earlier occurrence. These references, which specify the position and length of the earlier occurrence, and the remaining symbols are then encoded using an adaptive Huffman scheme [22, 26] to minimize the number of bits required to express them. This way, keywords and longer recurring sequences such as $\texttt{while}(\texttt{a} < \texttt{b})$ can be compressed down to just a few bits. As mentioned earlier, gzip compression of JavaScript and other files is so successful that many web servers and browsers provide

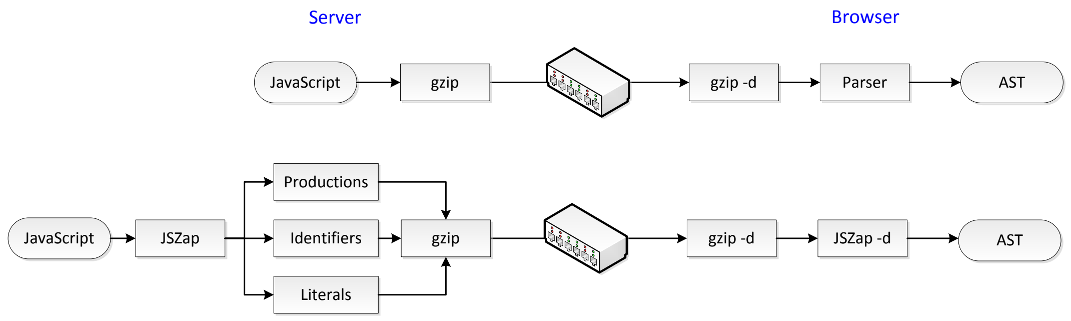
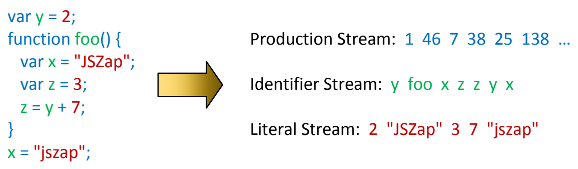


Figure 1: Architecture of JSZAP (bottom) compared to the current practice (top).

support for it, i.e., files are transparently compressed by the server and decompressed by the client. Nevertheless, gzip was not designed for JavaScript. Rather, it was designed as a general-purpose text compressor. Hence, it is possible to compress JavaScript even better with a special-purpose compressor like JSZAP that takes advantage of specific properties such as the structure imposed by the grammar.



```
var y = 2;
function foo() {
  var x = "JSZap";
  var z = 3;
  z = y + 7;
}
x = "jszap";
```

```
Production Stream:  1 46 7 38 25 138 ...

Identifier Stream:  y foo x z z y x

Literal Stream:  2 "JSZap" 3 7 "jszap"
```

Figure 2: A simple JavaScript example.

3.2 AST-based Compression

One way to expose the structure in JavaScript programs is to use a parser, which breaks down the source code into an abstract syntax tree (AST) whose nodes contain the tokens mentioned above. The AST specifies the order in which the grammar rules have to be applied to obtain the program at hand. In compiler terminology, these rules are called productions, the constants are called literals, and the variable and function names are called identifiers. Thus, the use of a parser in JSZAP makes it possible to extract and separate the productions, identifiers, and literals that represent a JavaScript program. Figure 1 illustrates this process, including further compression with gzip. The top portion of the figure shows the typical current process; the bottom portion illustrates the JSZAP approach of breaking down the code into multiple streams and compressing them separately.

Figure 2 provides an example of how a small piece of JavaScript code is converted into these three streams of data. The productions are shown in linearized format. The figure illustrates that the three categories exhibit very different properties, making it unlikely that a single compression algorithm will be able to compress all of them well. Instead, JSZAP applies different compression techniques to each category to maximize the compression ratio. Each compressor is designed to maximally exploit the characteristics of the corresponding category, as explained in the next section. Figure 3 shows that each category represents a significant fraction of the total amount of data, meaning that all three categories must be compressed well to obtain good overall compression. The figure shows results from our nine benchmarks, ordered in increasing size, ranging from 17 kilobytes to 668 kilobytes (see Figure 4). The fraction of each kind of data is consistent across the programs, with a slight trend that the larger files have a larger fraction of identifiers and a smaller fraction of literals.

4 JSZap Design and Implementation

Because data sent over the Internet are typically compressed with a general compression algorithm like gzip, we not only want to determine how to best compress ASTs but also how to do it in a way that complements this second compression stage well.

Such a two-stage compression scheme has interesting implications. For example, to optimize the overall compression ratio, it is not generally best to compress the data as much as possible in the first stage because doing so obfuscates patterns and makes it difficult for the second stage to be effective. In fact, the best approach may be to *expand* the amount of data in the first stage to better expose patterns and thereby making the second stage as useful as possible [9]. We now describe how JSZAP exploits the different properties of the productions, identifiers, and literals to compress them well in the presence of a gzip-based second compression stage.

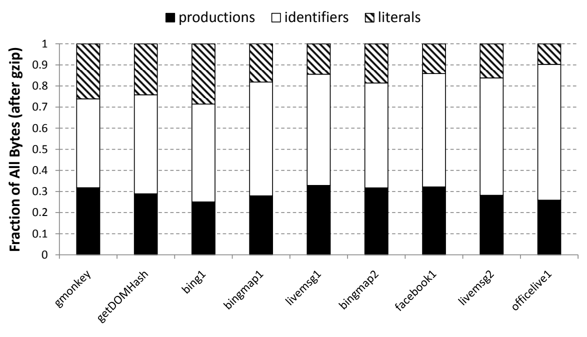


Figure 3: Breakdown of component types in our benchmarks (after gzip).

4.1 Compressing Productions

The JavaScript grammar used for JSZAP is a top-down LALR(k) grammar implemented using Visual Parse++ [41]. The grammar has 236 rules, which means that the linearized production stream consists of a sequence of bytes, each byte encoding a single grammar production. The productions can be compressed in tree or linearized format. We first attempted to compress the linearized form by employing renaming, using differential encoding, and removing chain productions.

Production renaming. Production renaming attempts to change the assignments of productions to integers (e.g., the production `Program => SourceElements` might be represented by the integer 225). We might choose to rename this production to integer 1 instead, if it was a common production in our streams. The idea behind renaming is to maximize the frequency of small production IDs. However, gzip is insensitive to absolute values as it only exploits repeating patterns, which is why this transformation does not help.

Differential encoding. Differential encoding works based on the observation that only a few productions can follow a given production. Hence, we renamed the following n productions to 0 through $n-1$ and did this for each production. For example, if production x can only be followed by the two productions with IDs 56 and 77, we would rename production 56 to 0 and production 77 to 1. Differential encoding can potentially help gzip because it reduces the number of unique bytes and increases their frequency, but it is unclear whether this results in longer or more frequent repeating patterns that gzip can exploit.

Chain rule. Some productions always follow one specific production. For such chains of productions, it suffices to record only the first production. While this approach substantially reduces the amount of data emitted, it does not help gzip because it only makes the repeating patterns shorter. Because gzip uses an adaptive Huffman coder to encode the lengths of the patterns, not much if anything is gained by this transformation. Moreover, differential encoding and chain production removal are antagonistic. By removing the chains, the number of symbols that can follow a specific symbol often increases.

Overall, the techniques we investigated to compress linearized production streams are not effective. Nevertheless, chain production removal is quite useful when compressing the productions in tree form, as the following subsection explains.

4.1.1 Compressing Productions in AST Format

We found that productions are more compressible in tree format. We believe the reason for this to be the following. Assume a production with two symbols on the right-hand-side, e.g., an `if` statement with a `then` and an `else` block. Such a production always corresponds to a node and its two children in the AST, no matter what context the production occurs in. In the linearized form, e.g., in a pre-order traversal of the tree, the first child appears right after the parent, but the second child appears at an arbitrary distance from the parent where the distance depends on the size of the subtree under the first child (the size of the `then` block in our example). This irregularity makes it difficult for any linear data model such as gzip's to anticipate the second symbol and therefore to achieve good compression.

Compressing the productions in tree form eliminates this problem. The children of a node can always be encoded in the context of the parent, making it easier to predict and compress the productions. The only additional piece of information needed is the position of the child since each child of a node has the same parent, grandparent, etc. In other words, we need to use the path from the root to a node as context for compressing that node plus information about which child it is. Without the position information, all children of a node would have the same context.

One very powerful context-based data compression technique is prediction by partial match (PPM) [11]. It works by recording, for each encountered context, what symbol follows so that the next time the same context is seen, a lookup can be performed to provide the likely next symbols together with their probability of occurring. The maximum allowed context length determines the size of the lookup table. We experimentally found a context length of one, i.e., just using the parent as well as the empty context, to yield the best performance after chain-production removal. Aside from maximizing the compression ratio, using only short contexts also greatly reduces the amount of memory needed for table space and makes decompression very fast, both of which are important when running on a constrained client such a a cell phone.

Since the algorithm may produce a different prediction for the order-0 context (a single table) and the order-1 context (one table per possible parent ID), we need to specify what to do if this happens. We use a PPM scheme that incorporates ideas from PPMA and PPMC [32], which have been shown to work well in practice. JSZAP's scheme always picks the longest context that has occurred at least once before, defaulting to the empty context if necessary. Because our tree nodes can have up to four children, JSZAP uses four distinct PPM tables, one for each position. For each context, the tables record how often each symbol follows. PPM then predicts the next symbol with a probability that is proportional to its frequency and uses an arithmetic coder [40] to compactly encode which symbol it actually is. This approach is so powerful that further compression with gzip is useless.

To ensure that each context can always make a prediction, the first-order contexts include an escape symbol, which is used to indicate that the current production has not been seen before and that the empty context needs to be queried. The frequency of the escape symbol is fixed at 1 (like in the PPMA method), which we found to work best. JSZAP primes the empty context with each possible production, which is to say that they are all initialized with a frequency of one. This way, no escape symbol is necessary. Unlike in conventional PPM implementations, where an order -1 context is used for this purpose, we opted to use the order 0 context because it tends to encounter most productions relatively quickly in any case.

To add aging, which gives more weight to recently seen productions, JSZAP scales down all frequency counts by a factor of two whenever one of the counts reaches a predefined maximum (as is done in the PPMC method). We found a maximum of 127 to work best. JSZAP further employs update exclusion, meaning that the empty context is not updated if the first-order context was able to predict the current production. Finally, and unlike most other PPM implementations, JSZAP does not need to encode an end-of-file symbol or record the length of the file because decompression automatically terminates when the tree is complete.

4.2 Compressing Identifiers

The identifiers are emitted in the order in which the parser encounters them. We considered several transformations to reduce the size of this identifier stream. First, the same identifiers are often used repeatedly. Second, some identifiers occur more often than others. Third, many identifier names are irrelevant.

Symbol tables. To exploit the fact that many identifiers appear frequently, JSZAP records each unique identifier in a symbol table and replaces the stream of identifiers by indices into this table. Per se, this transformation does not shrink the amount of data, but it enables the following optimizations.

At any one time, only a few identifiers are usually in scope. Hence, it is advantageous to split the symbol table into a global scope table and several local scope tables. Only one local scope table is active at a time, and function boundary information, which can be derived from the productions, is used to determine when to switch local scope tables. The benefit of this approach is that only a small number of indices are needed to specify the identifiers. Moreover, this approach enables several important additional optimizations. For instance, we can sort the global table by frequency to make small offsets more frequent.

Symbol table sorting. Because not all identifiers appear equally often, it pays to sort the symbol table from most to least frequently used identifier. As a result, the index stream contains mostly small values, which makes it more compressible when using variable-length encodings, which JSZAP does.

Local renaming. The actual names of local variables are irrelevant because JSZAP does not need to reproduce the variable names at the receiving end. One can rename local variables arbitrarily as long as uniqueness is guaranteed and there are no clashes with keywords or global identifiers. As mentioned before, one can give local variables very short names, such as `a`, `b`, `c`, etc., which is what many of the publicly available minifiers and JavaScript-generating compilers do.

Renaming allows JSZAP to use a built-in table of common variable names to eliminate the need to store the names explicitly. Consequently, most local scopes become empty and the index stream alone suffices to specify which identifier is used. (Essentially, the index is the variable name.) Note that renaming cannot be applied to global identifiers such as function names because external code may call these functions, which is done by name. Hence, JSZAP does not rename global identifiers.

4.3 Compressing Literals

The literals are also generated in the order in which the parser encounters them. The stream of literals contains three types of redundancy that we have tried to exploit. First, the same literal may occur multiple times. Second, there are different categories of literals such as strings, integers, and floating-point values. Third, some categories include known pre- and postfixes.

Symbol tables. We have attempted to take advantage of multiple occurrences of the same literal by storing all unique literals in a table and replacing the literal stream with a stream of indices into this table. Unfortunately, most literals occur only once. As a consequence, the index stream adds more overhead than the table of unique values saves, both with and without gzip compression. Thus, this approach expands instead of shrinks the amount of data.

Grouping literals by type. Exploiting the different categories of literals proved more fruitful, especially because the category can be determined from the productions, so no additional information needs to be recorded. JSZAP separates the string and numeric literals, which makes gzip more effective. For example, it is usually better to compress all strings and then all integer constants as opposed to compressing an interleaving of strings and integers.

Prefixes and postfixes. Eliminating known pre- and postfixes also aids the second compressor stage by not burdening it with having to compress unnecessary information and by transferring less data to it, which can make it faster and more effective because a larger fraction of the data fits into the search window [46]. The two optimizations JSZAP performs in this regard are removing the quotes around strings and using a single-character separator to delineate the literals instead of a newline, carriage-return pair. In practice, this optimization does not help much because gzip is largely insensitive to the length of repeating patterns.

5 Evaluation

In this section, we evaluate the performance of JSZAP using a variety of JavaScript source code taken from commercial web sites.

5.1 Experimental Setup

Figure 4 provides a summary of information about the benchmarks we have chosen to test JSZAP on. Each of the nine benchmarks is a JavaScript file, with its size indicated in the table both in terms of the number of bytes and lines of code, after pretty-printing (columns 2 and 3).

Many of the inputs come from online sources, including Bing, Bing Maps, Microsoft Live Messenger, and Microsoft Office Live. Two of the smaller scripts (`gmonkey`, `getDOMHash`) are hand-coded JavaScript applications used in browser plug-ins. The source files vary in size from 17 kilobytes to 668 kilobytes—results show that 100 kilobytes is not an uncommon size for JavaScript source in a high-function web application like Bing or Bing Maps [38].

We processed each input by running JSCrunch on the source code before compressing with gzip and JSZAP

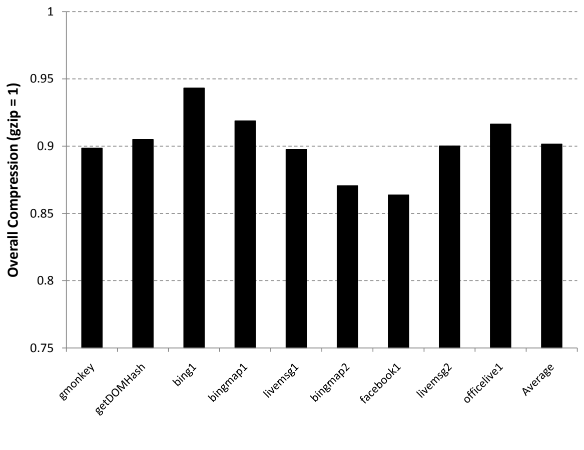


Figure 5: JSZAP overall compression relative to gzip.

(although in most cases this crunching had no effect because the code had already been crunched). We did not perform automatic local variable renaming with JSCrunch during our processing, because all but one of these files (`getDOMHash`) was received in a form that had been previous crunched with a tool such as JSCrunch or JSMin, meaning that the local variables had largely been renamed in the original source. Pretty-printing the sources results in files that range from 900 to 22,000 lines of code.

The size of the AST representation, composed of independent production, identifier, and literal streams, is shown in columns 4 and 5. Note that the AST is in fact already smaller than the corresponding JavaScript sources, by about 25% on average. This reduction results from elimination of source syntax such as keywords, delimiters, etc. The last two columns show the size of the gzipped representation and the gzip-to-source code ratio. In most cases, gzip compresses the source by a factor of 3–5x.

5.2 Total Compression

Figure 5 shows overall compression results of applying JSZAP to our benchmarks. We show the compression ratio of JSZAP compared to gzip. We see that in the majority of cases, the reduction in the overall size is 10% or more. It should be noted that JSZAP's AST-based representation already includes several processing steps such as parsing and semantic checking, thus reducing the amount of processing the client will have to do. Despite this fact, we are able to achieve compression ratios better than gzip. Another important factor to point out is that for a high-traffic host serving gigabytes of JavaScript to many thousands of users, savings of 10% may amount to hundreds of megabytes less to be sent.

Figure 5 demonstrates that the benefits of JSZAP com-

Benchmark name	Source bytes	Source lines	Uncompressed AST (bytes)	Uncompressed AST/src ratio	gzip bytes	gzip/source ratio
`gmonkey`	17,382	922	13,108	0.75	5,340	0.30
`getDOMHash`	25,467	1,136	17,462	0.68	6,908	0.27
`bing1`	77,891	3,758	65,301	0.83	23,454	0.30
`bingmap1`	80,066	3,473	56,045	0.69	19,537	0.24
`livemsg1`	93,982	5,307	70,628	0.75	22,257	0.23
`bingmap2`	113,393	9,726	108,082	0.95	41,844	0.36
`facebook1`	141,469	5,886	94,914	0.67	36,611	0.25
`livemsg2`	156,282	7,139	104,101	0.66	32,058	0.20
`officelive1`	668,051	22,016	447,122	0.66	132,289	0.19
Average				0.7432		0.2657

Figure 4: Summary of information about our benchmarks.

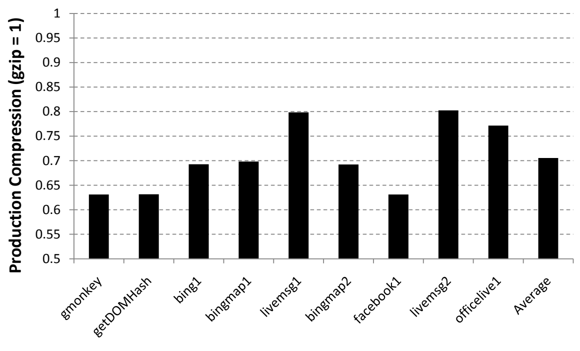


Figure 6: JSZAP production compression relative to gzip.

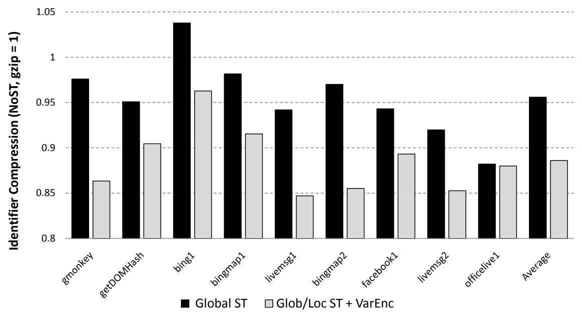


Figure 7: JSZAP identifier compression relative to gzip.

pression are largely independent of the input size. There is no clear correlation of compression ratios and whether the source has been produced by a tool or framework. This leads us to believe that similar compression benefits can be obtained with JSZAP for a wide range of JavaScript sources. The input with the greatest compression, `facebook1`, is also the input with the most effective compression of the productions relative to gzip (see next section), suggesting that the PPM compression of productions is a central part of an effective overall compression strategy.

5.3 Productions

Figure 6 shows the benefits of using PPM to independently compress the production stream. As we have discussed, the structured nature of the productions allows PPM compression to be very effective, producing a significant advantage over gzip. Just as before, we normalize the size produced using PPM compression relative to compressing the productions with gzip. We see that JSZAP compresses the productions 20% to over 35% better than gzip, with an average improvement of 30%. Again, JSZAP's compression benefits appear to be independent of the benchmark size.

We note that PPM compression can easily be changed with a number of control parameters. The results reported here are based on a context length of one and a maximum symbol frequency of 127. Varying these as well as other parameters slightly resulted in minor differences in overall compression, with individual file sizes changing by a few percent.

We note that our productions are particularly easy for gzip to compress because our JavaScript grammar contains less than 256 productions and thus each element of the productions file is a single byte. If the productions had required more than eight bits, gzip may have had less success in compressing the files whereas JSZAP would have largely remained unaffected.

5.4 Identifiers

Figure 7 presents the results of applying JSZAP to compress the identifier stream. The figure shows results normalized to using gzip to compress the identifier stream

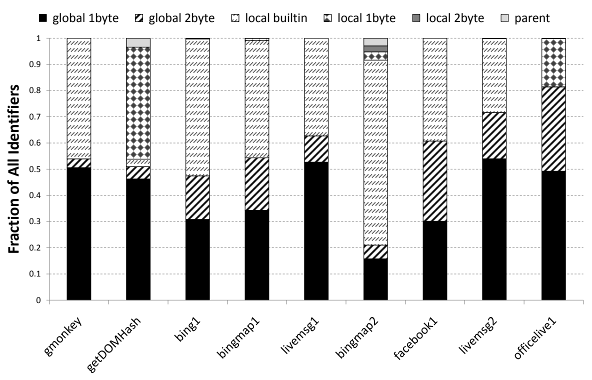


Figure 8: Breakdown of categories in variable-length identifier encoding.

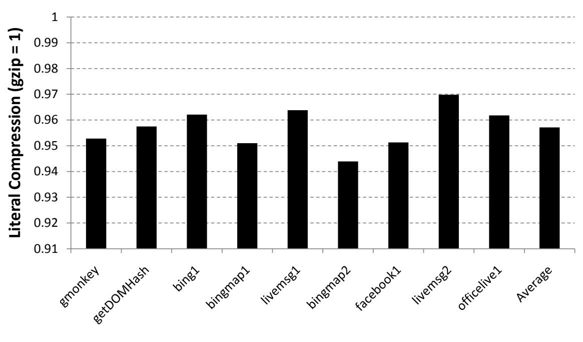


Figure 9: JSZAP literal compression relative to gzip.

without any symbol table being used. The figure includes two different symbol table encodings: a single global symbol table with a fixed-length 2-byte encoding (Global ST) as well as using both global and local symbol tables with variable-length encoding (Glob/Loc ST + VarEnc), as described in Section 4.2. We observe that the 2-byte encoding generally produces only small benefits and in one case hurts overall compression. Using the variable encoding results in a 12% improvement overall. Interestingly, we see that our variable encoding provides no benefit over the global symbol table in our largest application, `officelive1`.

To further illustrate the effectiveness of our variable-length encoding, Figure 8 shows a breakdown of different encoding types for our benchmarks. From the figure, we see that our strategy to represent as many identifiers as possible with 1 byte succeeded, with the only major category of 2-byte identifiers being globals. We see that global 1- and 2-byte identifiers account for more than half the total identifiers in most applications. Local built-ins are also very common in the applications, especially for `bingmap2`, where they account for over 75% of all identifiers. `bingmap2` is also one of the applications where we get the greatest benefit relative to gzip in Figure 7. Figure 8 explains why `officelive1` does not benefit from our variable-length encoding in the previous figure. Because of the framework that was used to generate `officelive1`, we see that more than 80% of all identifiers are globals, and there are no local built-ins. We anticipate that if we tailored variable renaming appropriately by preprocessing `officelive1`, we could replace many of the local 1-byte identifiers with built-ins. `getDOMHash`, which was written by hand and did not have automatic variable renaming performed during crunching, also has many 1-byte local variables.

To conclude, we see that we obtain a relatively modest compression (12%) of the identifiers over gzip, but because gzip is designed explicitly to compress characters strings, this result is not surprising. We do see that tailoring identifier compression using source-level information about local and global symbols does result in useful improvements.

5.5 Literals

Figure 9 shows the results of using JSZAP to process the literals before compressing them with gzip. As with the previous figures, we compare against compressing an unoptimized literal stream with gzip. Because literals are mostly single-use (except common ones such as 0, 1, etc.), using a literal table increases space usage over gzip and is not shown. Our other optimizations, including grouping literals by type and eliminating prefixes and postfixes have modest benefits, averaging around 4–5%.

6 Related Work

This section discusses compression in general and the specific work related to compressing JavaScript.

6.1 Mobile Code Compression

The work most closely related to JSZAP is the Slim Binaries and TransPROse projects by Franz et al. [4, 25]. Slim Binaries is the first project to promote the use of transmitting mobile code using abstract syntax trees for benefits including compression, code generation, and security. The original Slim Binaries project compressed Oberon ASTs using a variation of LZW compression [45] extended with abstract grammars. Later, in the TransPROse project [4, 24, 43], Franz et al. describe a new compression technique for ASTs based on abstract grammars, arithmetic coding, and prediction by partial match. They apply their technique to Java class files and compare the results against Pugh's highly-effective jar file compressor [36], as well as general compressors like gzip and bzip2. With PPM compression, they achieve a compression ratio of approximately 15% over the uncompressed source.

Our work is motivated by the goals of Franz's earlier work. We also share some of the techniques with that work, including PPM compression of the production rules. Our work differs in that we apply these ideas to compressing JavaScript. We show that for real JavaScript source code, we achieve significant benefits over the current state-of-the-art. In addition, we show that identifiers and literals constitute a significant fraction of the data that requires compression, and describe JavaScript-specific techniques to compress these streams.

In a one-page abstract, Evans describes the compression of Java and Pascal programs based on *guided parsing*, which also uses the language grammar to make compression more efficient [20]. In another one-page abstract, Eck et al. propose Syntax-oriented Coding [17]. Guided parsing and SoC have many elements in common with Slim Binaries and TransPROse, but due to their shortness, both papers lack detail and do not cite previous related work on Pascal.

Other approaches to compressing mobile code have been proposed. Many proposals have focused on compressing a program representation that is close to the target language, specifically native machine code [19] or some form of bytecode [21, 31]. Some proposals consider dynamically compressing unused portions of code to reduce the in-memory footprint of the executing program [15]. The main difference between this work and ours is our focus on using the augmented AST as the medium of transfer between the server and client as well as our focus on compressing the tree itself instead of a more linear format, such as an instruction stream.

Pugh considers ways to compress Java class files. He splits data into multiple streams using redundancies in the class file information and finds a number of format specific opportunities to achieve good compression [36]. Like our work, he examines opportunities to improve second-stage gzip compression, although he does not consider using the grammar to compress the program text. Jazz [8] and Clazz [27] also improve the representation of the entire Java archive but do not consider source compression.

6.2 Syntax-based Compression

The problem of compressing source code has been considered since the 1980s. The idea of using the program parse as a program representation and the grammar as a means of compressing the parse was proposed by Contla [12]. He applied the approach to Pascal source code and demonstrated compression on three small programs. Katajainen et al. describe a source program compressor for Pascal that encodes the parse tree and symbol tables [29]. They show that their Prolog implementation of the compressor results in space gains of 50–60%. Stone describes *analytic encoding*, which combines parsing with compression [42]. Stone considers how the parser used (LL versus LR) affects the resulting compressibility and reports that LR parsers are more appropriate, which is what JSZAP uses.

Cameron describes source compression using the language syntax as the data model [10]. He suggests using arithmetic coding to compress the production rules and separating the local and global symbols to improve the compression of symbols. In applying the technique to Pascal programs, he shows a result that is approximately 15% of the size of the original source. Tarhio reports the use of PPM compression on parse trees [44]. Applying the approach to four Pascal programs, he shows that the the number of bits per production can be reduced below more general purpose techniques such as gzip and bzip2.

Rai and Shankar consider compressing program intermediate code using tree compression [37]. They consider using tree grammars to encode the intermediate form (unlike work based on the source syntax) and show that they outperform gzip and bzip2 on lcc-generated intermediate files. They speculate that their technique could be applied to compression of XML-structured documents as well.

6.3 XML / Semistructured Text Compression

Adiego et al. describe LZCS [1, 2], a Lempel-Ziv-based algorithm to compress structured data such as XML files. Their approach takes advantage of repeated substructures by replacing them with a backward reference to an earlier occurrence. LZCS provides good compression and retains the ability to navigate and randomly access the data. When combined with a second compression stage such as PPM, the overall compression ratio is higher than when directly compressing the original data. JSZAP employs the same general approach; it also transforms the original data and uses a second compression stage to maximize the overall compression ratio.

The same authors further describe the Structural Contexts Model (SCM) [3], which exploits structural information such as XML tags to combine data belonging to the same structure. The combined data are more compressible than the original data because combining brings data with similar properties together. In contrast, elements from different structural units may be interleaved in the original data, which complicates compression. The authors use a separate PPM model for compressing each structure in SCM and propose a heuristic to combine structures whose properties are similar to further improve the compression ratio. JSZAP adopts the idea of separately compressing data with similar properties, i.e., identifiers, literals, and productions, to boost the compression ratio.

6.4 General Text and Other Compression

Bell et al. wrote a classic textbook that covers text compression in great detail [7]. It explains various compression algorithms, including Lempel-Ziv coding (used in gzip), prediction by partial match (PPM), and arithmetic coding. Moreover, it includes C source code for several compression algorithms. A survey by the same authors [6] on adaptive modeling for text compression provides additional information on the different PPM methods. Moffat et al. describe several important improvements to arithmetic coding that make it faster and easier to implement [33].

Arnold and Bell present a corpus for the evaluation of lossless compression algorithms [5], which contains some source code, though not JavaScript. The authors found that, while no corpus is likely to result in absolute compression ratios that are general, the relative performance of different compression algorithms is quite stable. Hence, we believe it is likely that JSZAP outperforms gzip not only on the inputs we measured but on most inputs.

Nevill-Manning and Witten designed an algorithm called Sequitur that identifies hierarchical structure in the input and uses it for compression [34]. The algorithm works by incrementally generating a context-free grammar that describes the input. Interestingly, Sequitur compresses well and produces a meaningful description of the structure in the input at the same time. Since JSZAP compresses the productions using the JavaScript grammar, it exploits the same type of structure as Sequitur does but without the need to dynamically derive the grammar.

7 Conclusions

This paper advocates an AST-based representation for delivering JavaScript code over the Internet and presents JSZAP, a tool that employs such an AST-based representation for compression. JSZAP compresses JavaScript code 10% better than gzip, which is the current standard. In the context of a high-traffic host serving gigabytes of JavaScript to thousands of users, the savings demonstrated by JSZAP may amount to hundreds of megabytes less to be transmitted. It is our hope that our work will inspire developers of next-generation browsers to reexamine their approach to representing, transmitting, and executing JavaScript code.

References

[1] J. Adiego, G. Navarro, and P. de la Fuente. Lempel-Ziv compression of structured text. pages 112–121, March 2004.

[2] J. Adiego, G. Navarro, and P. de la Fuente. Lempel-Ziv compression of highly structured documents: Research articles. *J. Am. Soc. Inf. Sci. Technol.*, 58(4):461–478, 2007.

[3] J. Adiego, G. Navarro, and P. de la Fuente. Using structural contexts to compress semistructured text collections. *Information Processing & Management*, 43(3):769–790, 2007. Special Issue on Heterogeneous and Distributed IR.

[4] W. Amme, P. S. Housel, N. Dalton, J. V. Ronne, P. H. Frohlich, C. H. Stork, V. Haldar, S. Zhenochin, and M. Franz. Project TRANSPROSE: Reconciling mobile-code security with execution efficiency. 2001.

[5] R. Arnold and T. Bell. A corpus for the evaluation of lossless compression algorithms. pages 201–210, 1997.

[6] T. Bell, I. H. Witten, and J. G. Cleary. Modeling for text compression. *ACM Comput. Surv.*, 21(4):557–591, 1989.

[7] T. C. Bell, J. G. Cleary, and I. H. Witten. *Text Compression*. Prentice Hall PTR, Englewood Cliffs, NJ, USA, 1990.

[8] Q. Bradley, R. N. Horspool, and J. Vitek. JAZZ: an efficient compressed format for Java archive files. In S. A. MacKay and J. H. Johnson, editors, *Proceedings of the 1998 conference of the Centre for Advanced Studies on Collaborative Research*, page 7. IBM, 1998.

[9] M. Burtscher, I. Ganusov, S. Jackson, J. Ke, P. Ratanaworabhan, and N. Sam. The VPC trace-compression algorithms. *IEEE Transactions on Computers*, 54(11):1329–1344, Nov. 2005.

[10] R. Cameron. Source encoding using syntactic information source models. *IEEE Transactions on Information Theory*, 34(4):843–850, Jul 1988.

[11] J. G. Cleary and I. H. Witten. Data compression using adaptive coding and partial string matching. *IEEE Transactions on Communications*, 32(4):396–402, 1984.

[12] J. F. Contla. Compact coding of syntactically

correct source programs. *Softw. Pract. Exper.*, 15(7):625–636, 1985.

[13] D. Crockford. ADsafe. adsafe.org.

[14] D. Crockford. JSMin: The JavaScript minifier. `http://www.crockford.com/javascript/jsmin.html`.

[15] S. Debray and W. Evans. Profile-guided code compression. In *Proc. Conf. on Programming Languages Design and Implementation*, June 2002.

[16] L. P. Deutsch. GZIP file format specification version 4.3. Internet RFC 1952, May 1996.

[17] P. Eck, X. Changsong, and R. Matzner. A new compression scheme for syntactically structured messages (programs) and its application to Java and the Internet. pages 542–, 1998.

[18] ECMA. ECMAScript language specification.

[19] J. Ernst, W. S. Evans, C. W. Fraser, S. Lucco, and T. A. Proebsting. Code compression. In *PLDI*, pages 358–365, 1997.

[20] W. S. Evans. Compression via guided parsing. In *Proceedings of the Conference on Data Compression*, page 544, Washington, DC, USA, 1998. IEEE Computer Society.

[21] W. S. Evans and C. W. Fraser. Bytecode compression via profiled grammar rewriting. In *PLDI*, pages 148–155, 2001.

[22] N. Faller. An adaptive system for data compression. In *Record of the 7th Asilomar Conference on Circuits, Systems, and Computers*, pages 593–597, 1973.

[23] C. Foster. JSCrunch: JavaScript cruncher. `http://www.cfoster.net/jscrunch/`.

[24] M. Franz, W. Amme, M. Beers, N. Dalton, P. H. Frohlich, V. Haldar, A. Hartmann, P. S. Housel, F. Reig, J. Ronne, C. H. Stork, and S. Zhenochin. Making mobile code both safe and efficient. In *Foundations of Intrusion Tolerant Systems*, Dec. 2002.

[25] M. Franz and T. Kistler. Slim binaries. *Commun. ACM*, 40(12):87–94, 1997.

[26] R. Gallager. Variations on a theme by Huffman. *IEEE Transactions on Information Theory*, 24(6):668–674, 1978.

[27] R. N. Horspool and J. Corless. Tailored compression of Java class files. *Softw, Pract. Exper*, 28(12):1253–1268, 1998.

[28] C. G. Jones, R. Liu, L. Meyerovich, K. Asanovic, and R. Bodk. Parallelizing the Web Browser. In *Proceedings of the Workshop on Hot Topics in Parallelism*. USENIX, March 2009.

[29] J. Katajainen, M. Penttonen, and J. Teuhola. Syntax-directed compression of program files. *Softw. Pract. Exper.*, 16(3):269–276, 1986.

[30] E. Kıcıman and B. Livshits. AjaxScope: a Platform for Remotely Monitoring the Client-Side Behavior of Web 2.0 Applications. In *Proceedings of Symposium on Operating Systems Principles*, Oct. 2007.

[31] S. Lucco. Split-stream dictionary program compression. In *Proc. Conf. on Programming Languages Design and Implementation*, pages 27–34, 2000.

[32] A. Moffat. Implementing the PPM data compression scheme. *IEEE Transactions on Communications*, 38(11):1917–1921, 1990.

[33] A. Moffat, R. M. Neal, and I. H. Witten. Arithmetic coding revisited. *ACM Trans. Inf. Syst.*, 16(3):256–294, 1998.

[34] C. G. Nevill-Manning and I. H. Witten. Compression and explanation using hierarchical grammars. *The Computer Journal*, 40(2/3):103–116, 1997.

[35] Pew Internet and American Project. Home broadband adoption 2007. `http://www.pewinternet.org/pdfs/PIP_Broadband\%202007.pdf`, 2007.

[36] W. Pugh. Compressing Java class files. In *PLDI*, pages 247–258, 1999.

[37] S. Rai and P. Shankar. Efficient statistical modeling for the compression of tree structured intermediate code. *Comput. J*, 46(5):476–486, 2003.

[38] P. Ratanaworabhan, B. Livshits, D. Simmons, and B. Zorn. JSMeter: Characterizing real-world behavior of JavaScript programs. Technical Report MSR-TR-2002-138, Microsoft Research, Microsoft Corporation, 2009.

[39] C. Reis, S. D. Gribble, T. Kohno, and N. C. Weaver. Detecting in-flight page changes with web tripwires. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implemen-*

tation, pages 31–44, 2008.

[40] J. J. Rissanen and G. G. Langdon. Arithmetic coding. *IBM Journal of Research and Development*, 23(2):149–162, 1979.

[41] Sandstone Technologies Inc. Parsing with Sandstone's Visual Parse++. `http://visualparse.s3.amazonaws.com/pvpp-fix.pdf`, 2001.

[42] R. G. Stone. On the choice of grammar and parser for the compact analytical encoding of programs. *The Computer Journal*, 29(5):307–314, 1986.

[43] C. H. Stork, V. Haldar, and M. Franz. Generic adaptive syntax-directed compression for mobile code. Technical Report 00-42, Department of Infomation and Computer Science, University of California, Irvine, 2000.

[44] J. Tarhio. On compression of parse trees. pages 205–211, Nov. 2001.

[45] T. A. Welch. A technique for high-performance data compression. *IEEE Computer*, 17(6):8–19, 1984.

[46] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, 23(3):337–343, 1977.