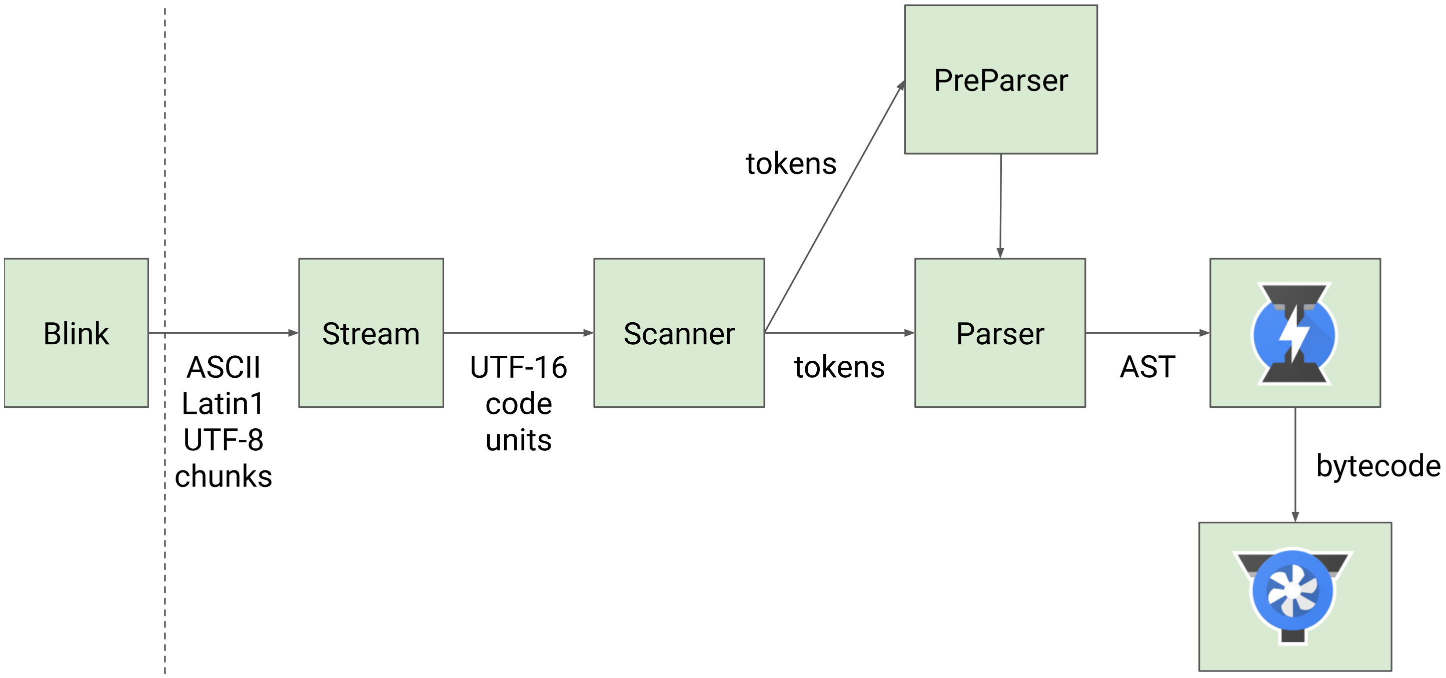
# Blazingly fast parsing, part 1: optimizing the scanner

Published 25 March 2019 · Tagged with internals parsing

To run a JavaScript program, the source text needs to be processed so V8 can understand it. V8 starts out by parsing the source into an abstract syntax tree (AST), a set of objects that represent the program structure. That AST gets compiled to bytecode by Ignition. The performance of these parse + compile phases is important: V8 cannot run code before compilation is done. In this series of blog posts, we focus on parsing, and the work done in V8 to ship a blazingly fast parser.

In fact, we start the series one stage before the parser. V8's parser consumes 'tokens' provided by the 'scanner'. Tokens are blocks of one or more characters that have a single semantic meaning: a string, an identifier, an operator like `++`. The scanner constructs these tokens by combining consecutive characters in an underlying character stream.

The scanner consumes a stream of Unicode characters. These Unicode characters are always decoded from a stream of UTF-16 code units. Only a single encoding is supported to avoid branching or specializing the scanner and parser for various encodings, and we chose UTF-16 since that's the encoding of JavaScript strings, and source positions need to be provided relative to that encoding. The `UTF16CharacterStream` provides a (possibly buffered) UTF-16 view over the underlying Latin1, UTF-8, or UTF-16 encoding that V8 receives from Chrome, which Chrome in turn received from the network. In addition to supporting more than one encoding, the separation between scanner and character stream allows V8 to transparently scan as if the entire source is available, even though we may only have received a portion of the data over the network so far.



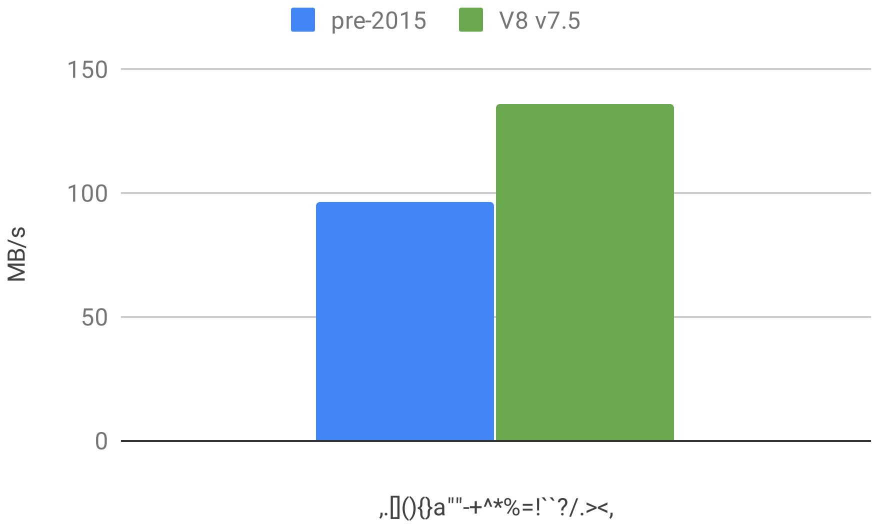
The interface between the scanner and the character stream is a method named `Utf16CharacterStream::Advance()` that returns either the next UTF-16 code unit, or `-1` to flag end of input. UTF-16 cannot encode every Unicode character in a single code unit. Characters outside the Basic Multilingual Plane are encoded as two code units, also called surrogate pairs. The scanner operates on Unicode characters rather than UTF-16 code units though, so it wraps this low-level stream interface in a `Scanner::Advance()` method that decodes UTF-16 code units into full Unicode characters. The currently decoded character is buffered and picked up by scan methods, such as `Scanner::ScanString()`.

The scanner chooses a specific scanner method or token based on a maximum lookahead of 4 characters, the longest ambiguous sequence of characters in JavaScript[1]. Once a method like `ScanString` is chosen, it consumes the remainder of characters for that token, buffering the first character that's not part of the token for the next scanned token. In the case of `ScanString` it also copies the scanned characters into a buffer encoded as Latin1 or UTF-16, while decoding escape sequences.

## Whitespace

Tokens can be separated by various types of whitespace, e.g., newline, space, tab, single line comments, multiline comments, etc. One type of whitespace can be followed by other types of whitespace. Whitespace adds meaning if it causes a line break between two tokens: that possibly results in automatic semicolon insertion. So before scanning the next token, all whitespace is skipped keeping track of whether a newline occured. Most real-world production JavaScript code is minified, and so multi-character whitespace luckily isn't very common. For that reason V8 uniformly scans each type of whitespace independently as if they were regular tokens. E.g., if the first token character is `/` followed by another `/`, V8 scans this as a single-line comment which returns `Token::WHITESPACE`. That loop simply continues scanning tokens until we find a token other than `Token::WHITESPACE`. This means that if the next token is not preceded by whitespace, we immediately start scanning the relevant token without needing to explicitly check for whitespace.

The loop itself however adds overhead to each scanned token: it requires a branch to verify the token that we've just scanned. It would be better to continue the loop only if the token we have just scanned could be a `Token::WHITESPACE`. Otherwise we should just break out of the loop. We do this by moving the loop itself into a separate helper method from which we return immediately when we're certain the token isn't `Token::WHITESPACE`. Even though these kinds of changes may seem really small, they remove overhead for each scanned token. This especially makes a difference for really short tokens like punctuation:

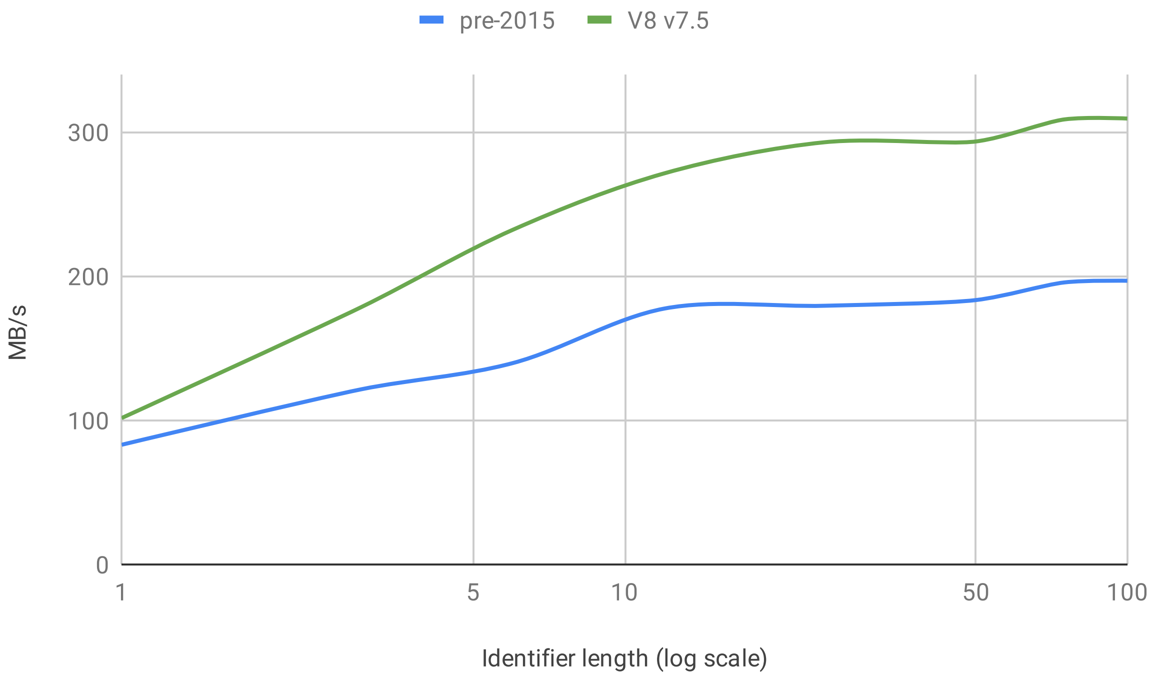


## Identifier scanning

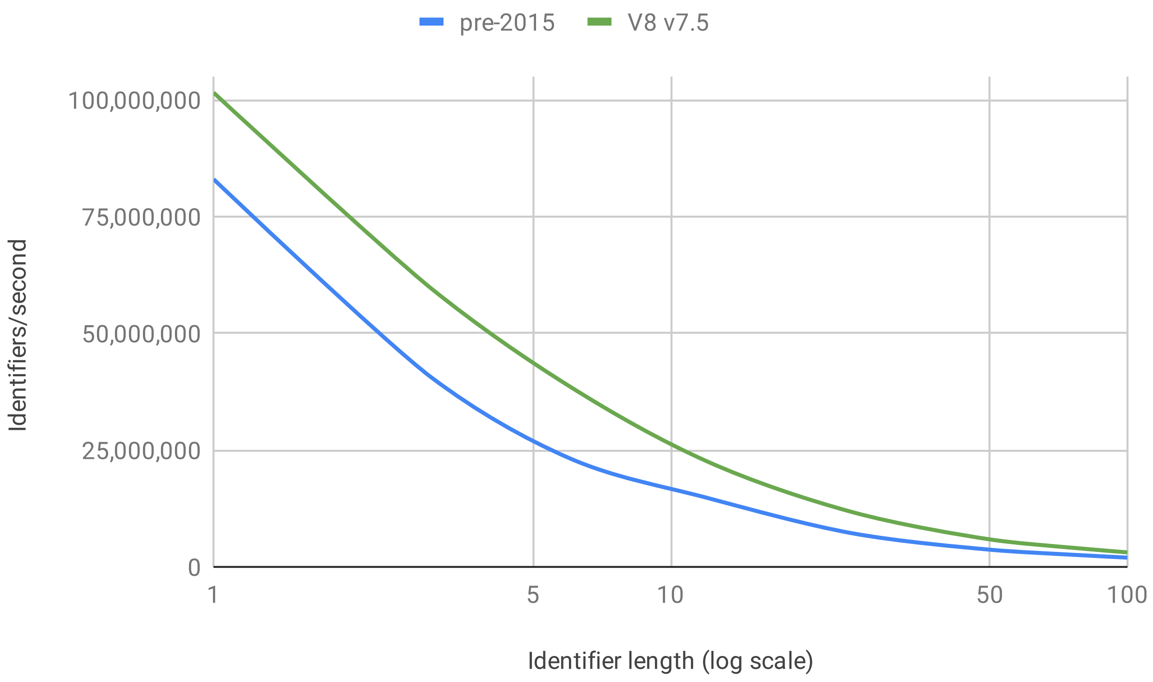
The most complicated, but also most common token, is the identifier token, which is used for variable names (among other things) in JavaScript. Identifiers start with a Unicode character with the property `ID_Start`, optionally followed by a sequence of characters with the property `ID_Continue`. Looking up whether a Unicode character has the property `ID_Start` or `ID_Continue` is quite expensive. By inserting a cache mapping from characters to their properties we can speed this up a bit.

Most JavaScript source code is written using ASCII characters though. Of the ASCII-range characters, only `a-z`, `A-Z`, `$` and `_` are identifier start characters. `ID_Continue` additionally includes `0-9`. We speed up identifier scanning by building a table with flags for each of the 128 ASCII characters indicating whether the character is an `ID_Start`, an `ID_Continue` character, etc. While characters we’re looking at are within ASCII range, we look up the respective flags in this table and verify a property with a single branch. Characters are part of the identifier until we see the first character that does not have the `ID_Continue` property.

All the improvements mentioned in this post add up to the following difference in identifier scanning performance:



It may seem counterintuitive that longer identifiers scan faster. That might make you think that it’s beneficial for performance to increase the identifier length. Scanning longer identifiers is simply faster in terms of MB/s because we stay longer in a very tight loop without returning to the parser. What you care about from the point-of-view of the performance of your application, however, is how fast we can scan full tokens. The following graph roughly shows the number of tokens we scan per second relative to the token length:



Here it becomes clear that using shorter identifiers is beneficial for the parse performance of your application: we're able to scan more tokens per second. This means that sites that we seem to parse faster in MB/s simply have lower information density, and actually produce fewer tokens per second.

## Internalizing minified identifiers

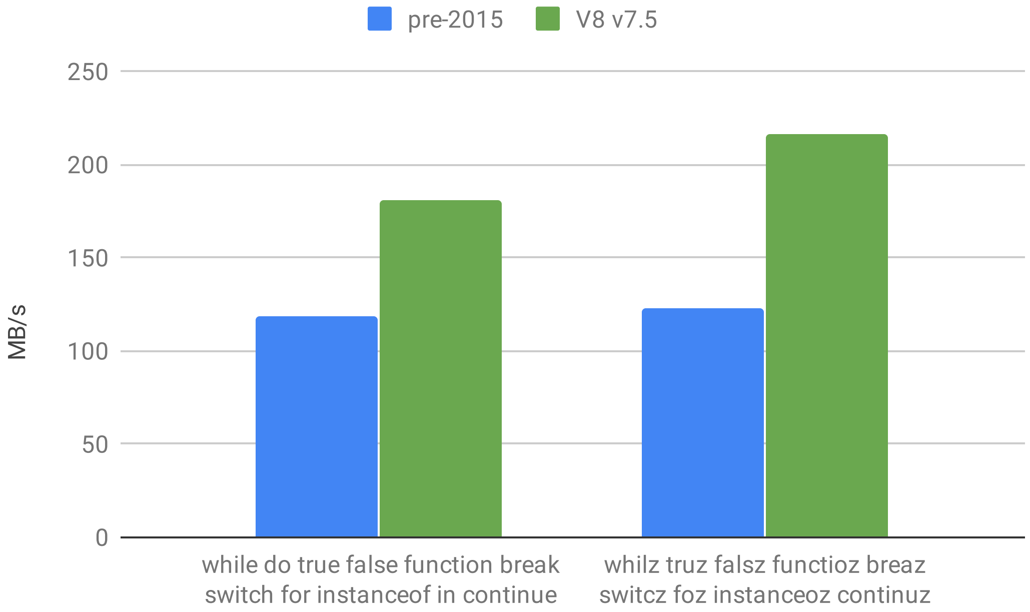
All string literals and identifiers are deduplicated on the boundary between the scanner and the parser. If the parser requests the value of a string or identifier, it receives a unique string object for each possible literal value. This typically requires a hash table lookup. Since JavaScript code is often minified, V8 uses a simple lookup table for single ASCII character strings.

## Keywords

Keywords are a special subset of identifiers defined by the language, e.g., `if`, `else`, and `function`. V8's scanner returns different tokens for keywords than for identifiers. After scanning an identifier we need to recognize whether the identifier is a keyword. Since all keywords in JavaScript only contain lowercase characters `a-z`, we also keep flags indicating whether ASCII characters are possible keyword start and continue characters.

If an identifier can be a keyword according to the flags, we could find a subset of keyword candidates by switching over the first character of the identifier. There are more distinct first characters than lengths of keywords, so it reduces the number of subsequent branches. For each character, we branch based on the possible keyword lengths and only compare the identifier with the keyword if the length matches as well.

Better is to use a technique called perfect hashing. Since the list of keywords is static, we can compute a perfect hash function that for each identifier gives us at most one candidate keyword. V8 uses gperf to compute this function. The result computes a hash from the length and first two identifier characters to find the single candidate keyword. We only compare the identifier with the keyword if the length of that keyword matches the input identifier length. This especially speeds up the case where an identifier isn't a keyword since we need fewer branches to figure it out.

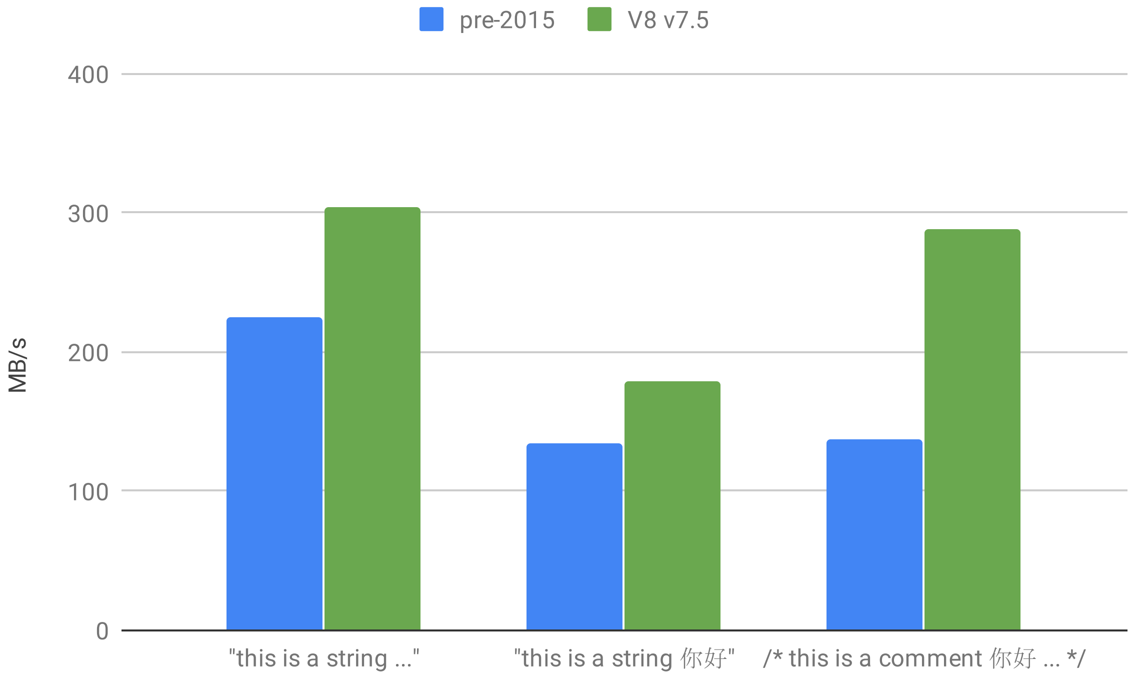


## Surrogate pairs

As mentioned earlier, our scanner operates on a UTF-16 encoded stream of characters, but consumes Unicode characters. Characters in supplementary planes only have a special meaning for identifier tokens. If for example such characters occur in a string, they do not terminate the string. Lone surrogates are supported by JS and are simply copied from the source as well. For that reason it is better to avoid combining surrogate pairs until absolutely necessary, and let the scanner operate directly on UTF-16 code units instead of Unicode characters. When we are scanning a string, we do not need to look for surrogate pairs, combine them, and then later split them again when we stash away the characters to build up a literal. There are only two remaining places where the scanner does need to deal with surrogate pairs. At the start of token scanning, only when we don't recognize a character as anything else do we need to combine surrogate pairs to check whether the result is an identifier start. Similarly, we need to combine surrogate pairs in the slow path of identifier scanning dealing with non-ASCII characters.

## `AdvanceUntil`

The interface between the scanner and the `UTF16CharacterStream` makes the boundary quite stateful. The stream keeps track of its position in the buffer, which it increments after each consumed code unit. The scanner buffers a received code unit before going back to the scan method that requested the character. That method reads the buffered character and continues based on its value. This provides nice layering, but is fairly slow. Last fall, our intern Florian Sattler came up with an improved interface that keeps the benefits of the layering while providing much faster access to code units in the stream. A templatized function `AdvanceUntil`, specialized for a specific scan helper, calls the helper for each character in the stream until the helper returns false. This essentially provides the scanner direct access to the underlying data without breaking abstractions. It actually simplifies the scan helper functions since they do not need to deal with `EndOfInput`.



`AdvanceUntil` is especially useful to speed up scan functions that may need to consume large numbers of characters. We used it to speed up identifiers already shown earlier, but also strings[2] and comments.

## Conclusion

The performance of scanning is the cornerstone of parser performance. We've tweaked our scanner to be as efficient as possible. This resulted in improvements across the board, improving the performance of single token scanning by roughly 1.4×, string scanning by 1.3×, multiline comment scanning by 2.1×, and identifier scanning by 1.2–1.5× depending on the identifier length.

Our scanner can only do so much however. As a developer you can further improve parsing performance by increasing the information density of your programs. The easiest way to do so is by minifying your source code, stripping out unnecessary whitespace, and to avoid non-ASCII identifiers where possible. Ideally, these steps are automated as part of a build process, in which case you don't have to worry about it when authoring code.

---

1. `<!--` is the start of an HTML comment, whereas `<!-` scans as "less than", "not", "minus". ↩
2. Strings and identifiers that cannot be encoded in Latin1 are currently more expensive since we first try to buffer them as Latin1, converting them to UTF-16 once we encounter a character that cannot be encoded in Latin1. ↩

Posted by Toon Verwaest (@tverwaes), scandalous optimizer.

Retweet this article!