Frequently asked Questions

- What are these `rescanFooToken` functions in the scanner for?
- Why does the lexical classifier sometimes give inaccurate results for nested template strings?

==========

What are these `rescanFooToken` functions in the scanner for?

The ECMAScript grammar describes *lexical goals* for its grammar, for which an alternate scanning rule should be used in its place from the default. These rules are triggered when ***syntactically aware consumers*** require them (i.e. ECMAScript parsers which know when a construct can occur). For details, see the current ES spec.

One example of this is for a single `/` (the forward-slash token). As long as the `/` isn't immediately followed by another `/` (indicating a comment), the default goal (*InputElementDiv*) is to scan it as a plain `/` or `/=` (for division operations); however, in contexts where a bare `/` or `/=` would not make sense (such as when parsing a *PrimaryExpression*), the goal is modified to *InputElementRegExp* to scan out a regular expression literal.

The rescan functions roughly correspond to triggering the alternate rules, though rather than passing an extra parameter, a rescan is demanded of the scanner.

Though lexical goals are not addressed in the TypeScript spec, there is effectively one new rule added for the `>` s (greater-than characters) to make generics easy to parse, but impose that any places which might encounter a `>>`, `>>>`, etc. require a rescan.

==========

Why does the lexical classifier sometimes give inaccurate results for nested template strings?

Template strings did not even originally have lexical classification support prior to 1.5 for several technical reasons. However, due to demand, support was added for what we believed would be the majority of practical uses.

As a precursor, when discussing template expressions, it is useful to be familiar with the following syntactic components:

Syntax Kind	Example
`NoSubstitutionTemplateLiteral`	`` `Hello world` ``
`TemplateHead`	`` `abc ${ ``
`TemplateMiddle`	`} def ${`
`TemplateTail`	`} ghi ` ``
`TemplateExpression`	`` `abc ${ 0 } def ${ 1 } ghi ` ``

The lexical classifier is a rudimentary classifier that is intended to be fast and work on single lines at a time. It works through augmenting TypeScript's scanner, which itself works on a regular language grammar except when given a lexical goal in mind. Keep in mind, lexical goals can only be triggered accurately when motivated by a syntactically aware entity, while the lexical classifier only has the context of a single line with a previous line state to work with.

To throw a wrench in the gears, substitution templates are not regular (they are context-free, and if someone wants to write a formal proof of this, *a la* Rust's string literals not being context-free, feel free to send a pumping lemma pull request for the wiki). The issue is that a `}` (close curly brace) and the template tail literals (`}...${`

(*TemplateMiddle*) and `}...${` (*TemplateTail*)) are need to be distinguished by lexical goals, which are triggered by ***syntactically aware*** consumers.

The basic solution is to just maintain a stack. However, in the interest of giving accurate results without complicating the classifier too much, we only keep track of whether a *single* template expression was unterminated. This means that if you have `` `...${ `...${ `` (two *TemplateHead*s) on a single line, or a multiline object literal in substitution position, your results may not be accurate. In practice this does not happen much, so the behavior is largely acceptable.

In any case, consumers who need classifiers should rely on the syntactic (and semantic) classifiers if accuracy is important, using the lexical classifier as a fallback.