PEG.js

Parser Generator for JavaScript

Documentation

Table of Contents

- Installation
  - Node.js
  - Browser
- Generating a Parser
  - Command Line
  - JavaScript API

- Using the Parser
- Grammar Syntax and Semantics
    - Parsing Expression Types
- Compatibility

Installation

Node.js

To use the `pegjs` command, install PEG.js globally:

```
$ npm install -g pegjs
```

To use the JavaScript API, install PEG.js locally:

```
$ npm install pegjs
```

If you need both the `pegjs` command and the JavaScript API, install PEG.js both ways.

Browser

Download the PEG.js library (regular or minified version) or install it using Bower:

```
$ bower install pegjs
```

Generating a Parser

PEG.js generates parser from a grammar that describes expected input and can specify what the parser returns (using semantic actions on matched parts of the input). Generated parser itself is a JavaScript object with a simple API.

Command Line

To generate a parser from your grammar, use the `pegjs` command:

```
$ pegjs arithmetics.pegjs
```

This writes parser source code into a file with the same name as the grammar file but with ".js" extension. You can also specify the output file explicitly:

```
$ pegjs arithmetics.pegjs arithmetics-parser.js
```

If you omit both input and output file, standard input and output are used.

By default, the parser object is assigned to `module.exports`, which makes the output a Node.js module. You can assign it to another variable by passing a variable name using the `-e`/`--export-var` option. This may be helpful if you want to use the parser in browser environment.

You can tweak the generated parser with several options:

`--cache`

Makes the parser cache results, avoiding exponential parsing time in pathological cases but making the parser slower.

`--allowed-start-rules`

Comma-separated list of rules the parser will be allowed to start parsing from (default: the first rule in the grammar).

`--plugin`

Makes PEG.js use a specified plugin (can be specified multiple times).

`--extra-options`

Additional options (in JSON format) to pass to `PEG.buildParser`.

`--extra-options-file`

File with additional options (in JSON format) to pass to `PEG.buildParser`.

`--trace`

Makes the parser trace its progress.

JavaScript API

In Node.js, require the PEG.js parser generator module:

```
var PEG = require("pegjs");
```

In browser, include the PEG.js library in your web page or application using the `<script>` tag. The API will be available in the `PEG` global object.

To generate a parser, call the `PEG.buildParser` method and pass your grammar as a parameter:

```
var parser = PEG.buildParser("start = ('a' / 'b')+");
```

The method will return generated parser object or its source code as a string (depending on the value of the `output` option — see below). It will throw an exception if the grammar is invalid. The exception will contain `message` property with more details about the error.

You can tweak the generated parser by passing a second parameter with an options object to `PEG.buildParser`. The following options are supported:

`cache`

: If `true`, makes the parser cache results, avoiding exponential parsing time in pathological cases but making the parser slower (default: `false`).

`allowedStartRules`

: Rules the parser will be allowed to start parsing from (default: the first rule in the grammar).

`output`

: If set to `"parser"`, the method will return generated parser object; if set to "source", it will return parser source code as a string (default: `"parser"`).

`optimize`

Selects between optimizing the generated parser for parsing speed (`"speed"`) or code size (`"size"`) (default: `"speed"`).

`plugins`

Plugins to use.

Using the Parser

Using the generated parser is simple — just call its `parse` method and pass an input string as a parameter. The method will return a parse result (the exact value depends on the grammar used to build the parser) or throw an exception if the input is invalid. The exception will contain `location`, `expected`, `found` and `message` properties with more details about the error.

```
parser.parse("abba"); // returns ["a", "b", "b", "a"]

parser.parse("abcd"); // throws an exception
```

You can tweak parser behavior by passing a second parameter with an options object to the `parse` method. The following options are supported:

`startRule`

Name of the rule to start parsing from.

`tracer`

Tracer to use.

Parsers can also support their own custom options.

Grammar Syntax and Semantics

The grammar syntax is similar to JavaScript in that it is not line-oriented and ignores whitespace between tokens. You can also use JavaScript-style comments (`// ...` and `/* ... */`).

Let's look at example grammar that recognizes simple arithmetic expressions like `2*(3+4)`. A parser generated from this grammar computes their values.

```
start
  = additive

additive
  = left:multiplicative "+" right:additive { return left + right; }
  / multiplicative

multiplicative
  = left:primary "*" right:multiplicative { return left * right; }
  / primary

primary
  = integer
  / "(" additive:additive ")" { return additive; }

integer "integer"
  = digits:[0-9]+ { return parseInt(digits.join(""), 10); }
```

On the top level, the grammar consists of *rules* (in our example, there are five of them). Each rule has a *name* (e.g. `integer`) that identifies the rule, and a *parsing expression* (e.g. `digits:[0-9]+ { return parseInt(digits.join(""), 10); }`) that defines a pattern to match against the input text and possibly contains some JavaScript code that determines what happens when the pattern matches successfully. A rule can also contain *human-readable name* that is used in error messages (in our example, only the `integer` rule has a human-readable name). The parsing starts at the first rule, which is also called the *start rule*.

A rule name must be a JavaScript identifier. It is followed by an equality sign ("=") and a parsing expression. If the rule has a human-readable name, it is written as a JavaScript string between the name and separating equality sign. Rules need to be separated only by whitespace (their beginning is easily recognizable), but a semicolon (";") after the parsing expression is allowed.

The first rule can be preceded by an *initializer* — a piece of JavaScript code in curly braces ("{" and "}"). This code is executed before the generated parser starts parsing. All variables and functions defined in the initializer are accessible in rule actions and semantic predicates. The code inside the initializer can access the parser object using the `parser` variable and options passed to the parser using the `options` variable. Curly braces in the initializer code must be balanced. Let's look at the example grammar from above using a simple initializer.

```
{
  function makeInteger(o) {
    return parseInt(o.join(""), 10);
  }
}

start
  = additive
```

```
additive
  = left:multiplicative "+" right:additive { return left + right; }
  / multiplicative

multiplicative
  = left:primary "*" right:multiplicative { return left * right; }
  / primary

primary
  = integer
  / "(" additive:additive ")" { return additive; }

integer "integer"
  = digits:[0-9]+ { return makeInteger(digits); }
```

The parsing expressions of the rules are used to match the input text to the grammar. There are various types of expressions — matching characters or character classes, indicating optional parts and repetition, etc. Expressions can also contain references to other rules. See detailed description below.

If an expression successfully matches a part of the text when running the generated parser, it produces a *match result*, which is a JavaScript value. For example:

- An expression matching a literal string produces a JavaScript string containing matched part of the input.
- An expression matching repeated occurrence of some subexpression produces a JavaScript array with all the matches.

The match results propagate through the rules when the rule names are used in expressions, up to the start rule. The generated parser returns start rule's match result when parsing is successful.

One special case of parser expression is a *parser action* — a piece of JavaScript code inside curly braces ("{" and "}") that takes match results of some of the the preceding expressions and returns a JavaScript value. This value is considered match result of the preceding expression (in other words, the parser action is a match result transformer).

In our arithmetics example, there are many parser actions. Consider the action in expression `digits:[0-9]+ { return parseInt(digits.join(""), 10); }`. It takes the match result of the expression [0-9]+, which is an array of strings containing digits, as its parameter. It joins the digits together to form a number and converts it to a JavaScript `number` object.

Parsing Expression Types

There are several types of parsing expressions, some of them containing subexpressions and thus forming a recursive structure:

"*literal*"
'*literal*'

> Match exact literal string and return it. The string syntax is the same as in JavaScript. Appending `i` right after the literal makes the match case-insensitive.

.

> Match exactly one character and return it as a string.

[*characters*]

Match one character from a set and return it as a string. The characters in the list can be escaped in exactly the same way as in JavaScript string. The list of characters can also contain ranges (e.g. `[a-z]` means “all lowercase letters”). Preceding the characters with `^` inverts the matched set (e.g. `[^a-z]` means “all character but lowercase letters”). Appending `i` right after the literal makes the match case-insensitive.

rule

Match a parsing expression of a rule recursively and return its match result.

(*expression*)

Match a subexpression and return its match result.

***expression* ***

Match zero or more repetitions of the expression and return their match results in an array. The matching is greedy, i.e. the parser tries to match the expression as many times as possible. Unlike in regular expressions, there is no backtracking.

***expression* +**

Match one or more repetitions of the expression and return their match results in an array. The matching is greedy, i.e. the parser tries to match the expression as many times as possible. Unlike in regular expressions, there is no backtracking.

***expression* ?**

Try to match the expression. If the match succeeds, return its match result, otherwise return `null`.

Unlike in regular expressions, there is no backtracking.

`&` *`expression`*

Try to match the expression. If the match succeeds, just return `undefined` and do not advance the parser position, otherwise consider the match failed.

`!` *`expression`*

Try to match the expression. If the match does not succeed, just return `undefined` and do not advance the parser position, otherwise consider the match failed.

`& {` *`predicate`* `}`

The predicate is a piece of JavaScript code that is executed as if it was inside a function. It gets the match results of labeled expressions in preceding expression as its arguments. It should return some JavaScript value using the `return` statement. If the returned value evaluates to `true` in boolean context, just return `undefined` and do not advance the parser position; otherwise consider the match failed.

The code inside the predicate can access all variables and functions defined in the initializer at the beginning of the grammar.

The code inside the predicate can also access location information using the `location` function. It returns an object like this:

```
{
  start: { offset: 23, line: 5, column: 6 },
  end:   { offset: 23, line: 5, column: 6 }
}
```

The `start` and `end` properties both refer to the current parse position. The `offset` property contains an offset as a zero-based index and `line` and `column` properties contain a line and a column as one-based indices.

The code inside the predicate can also access the parser object using the `parser` variable and options passed to the parser using the `options` variable.

Note that curly braces in the predicate code must be balanced.

`! { predicate }`

The predicate is a piece of JavaScript code that is executed as if it was inside a function. It gets the match results of labeled expressions in preceding expression as its arguments. It should return some JavaScript value using the `return` statement. If the returned value evaluates to `false` in boolean context, just return `undefined` and do not advance the parser position; otherwise consider the match failed.

The code inside the predicate can access all variables and functions defined in the initializer at the beginning of the grammar.

The code inside the predicate can also access location information using the `location` function. It returns an object like this:

```
{
  start: { offset: 23, line: 5, column: 6 },
  end:   { offset: 23, line: 5, column: 6 }
}
```

The `start` and `end` properties both refer to the current parse position. The `offset` property contains an

offset as a zero-based index and `line` and `column` properties contain a line and a column as one-based indices.

The code inside the predicate can also access the parser object using the `parser` variable and options passed to the parser using the `options` variable.

Note that curly braces in the predicate code must be balanced.

$ *expression*

Try to match the expression. If the match succeeds, return the matched string instead of the match result.

label* : *expression

Match the expression and remember its match result under given label. The label must be a JavaScript identifier.

Labeled expressions are useful together with actions, where saved match results can be accessed by action's JavaScript code.

***expression*$_1$ *expression*$_2$ ... *expression*$_n$**

Match a sequence of expressions and return their match results in an array.

***expression* { *action* }**

Match the expression. If the match is successful, run the action, otherwise consider the match failed.

The action is a piece of JavaScript code that is executed as if it was inside a function. It gets the match results of labeled expressions in preceding expression as its arguments. The action should return some JavaScript value using the `return` statement. This value is considered match result of the preceding expression.

To indicate an error, the code inside the action can invoke the `expected` function, which makes the parser throw an exception. The function takes one parameter — a description of what was expected at the current position. This description will be used as part of a message of the thrown exception.

The code inside an action can also invoke the `error` function, which also makes the parser throw an exception. The function takes one parameter — an error message. This message will be used by the thrown exception.

The code inside the action can access all variables and functions defined in the initializer at the beginning of the grammar. Curly braces in the action code must be balanced.

The code inside the action can also access the string matched by the expression using the `text` function.

The code inside the action can also access location information using the `location` function. It returns an object like this:

```
{
  start: { offset: 23, line: 5, column: 6 },
  end:   { offset: 25, line: 5, column: 8 }
}
```

The `start` property refers to the position at the beginning of the expression, the `end` property refers to position after the end of the expression. The `offset` property contains an offset as a zero-based index

and `line` and `column` properties contain a line and a column as one-based indices.

The code inside the action can also access the parser object using the `parser` variable and options passed to the parser using the `options` variable.

Note that curly braces in the action code must be balanced.

***expression*$_1$ / *expression*$_2$ / ... / *expression*$_n$**

Try to match the first expression, if it does not succeed, try the second one, etc. Return the match result of the first successfully matched expression. If no expression matches, consider the match failed.

Compatibility

Both the parser generator and generated parsers should run well in the following environments:

- Node.js 0.10.0+
- io.js
- Internet Explorer 8+
- Edge
- Firefox
- Chrome
- Safari
- Opera