tsPEG documentation

A.1   tsPEG : A PEG Parser Generator for TypeScript

tsPEG is a PEG Parser generator for TypeScript. *tsPEG* takes in an intuitive description of a grammar and outputs a fully featured parser that takes full advantage of the TypeScript type system.

A.1.1   Installation

tspeg can be installed by running

```
npm install -g tspeg
```

A.1.2   Features

- Fully featured PEG support, more powerful than CFGs.

- Infinite lookahead parsing, no restrictions.

- Regex based lexing, implicit tokenisation in your grammar specification.

- Tight typing, generates classes for all production rules, differentiable using discriminated unions.

A.1.3   CLI Usage

The CLI invocation syntax is as follows

```
tspeg <grammar-file> <output-file>
```

This generates a TypeScript ES6 module that exports a parser class, as well as classes that represent your AST.

A.1.4   Parser Usage

The parser exports a parse function that accepts an input string, and returns ParseResult object like this

```
class ParseResult {
    ast: START | null;
    err: SyntaxErr | null;
}
```

If the err field is non-null, then a syntax error was found, otherwise the AST is stored in the ast field.

A.1.5   Grammar Syntax

tsPEG grammars are specified with a simple syntax, similar to the classic EBNF syntax.

Grammars are composed of a sequence of rules, each rule defines what text that it should match, using regex literals, names of rules, and powerful operators like | for choice.

Each rule is defined by a name, followed by a := sign and then a "rule description". Rule descriptions are easiest seen by example here:

```
hello := 'Hello '
helloWorld := hello 'World'
helloChoice := hello 'Mars' | helloWorld
```

- The first line defines a rule `hello` which matches the string 'Hello' directly.

- The next line defines the rule `helloWorld`, this rule first matches our first rule `hello`, then matches the string 'World', i.e it matches the string "Hello World".

- The third line uses the | operator to make a choice between two options.

    1. First this rule will try to match the left hand side of the operator: `hello 'Mars'`, which as before will first match the rule `hello`, then the regex literal `Mars`.

    2. If this left hand side fails, we move on to trying the right hand side, which is just a reference to the `helloWorld` rule. In practice this means it either matches "Hello World" or "Hello Mars"

tsPEG **starts parsing with the first rule in the grammar** so to match the `helloChoice` rule we should either move it to the start or defined a `start` rule that points to it like this:

```
start := helloChoice
hello := 'Hello '
helloWorld := hello 'World'
helloChoice := hello 'Mars' | helloWorld
```

Putting this grammar into a file called "grammar.peg" and running `tspeg grammar.peg parser.ts` we get our parser for this simple grammar in the file "parser.ts". We can compile this (target at least es2015) and load it into NodeJS to test it. This can be seen in Figure A.1.
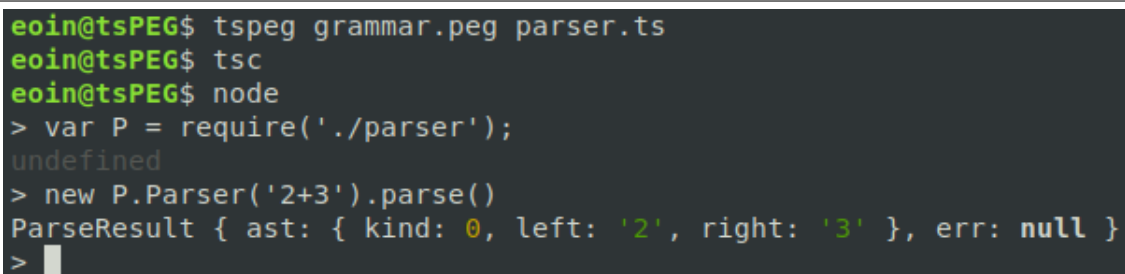

Figure A.1: Example 1

As you can see we get `null` error for 'Hello World', and 'Hello Mars', this means the match was successful. But when we try 'Hello Jupiter' we get an error object, and it lists the location of the error, and the expected matches at that location, namely 'Mars' or 'World'. You can see more about errors in the Syntax Error section.

Notably the `ast` field of the parse result is empty. This is because we haven't told *tsPEG* what elements of the grammar we would like to be returned. When we write a rule definition, we can specify the fields we'd like to save in the AST by assigning them with an = sign. For example, say we want to match a string that's the sum of 2 numbers, e.g. "2+3", "123+456", we'd like to save both side of the sum in our AST, we can do this by writing a grammar like the following. *Note that we had to write '\+' to escape the '+' symbol as the '+' symbol has special meaning in regex.*

```
sum := left=num '\+' right=num
num := '[0-9]+'
```

Running *tsPEG* on this input and testing the parser in node we can see that the parser result has saved the left and right numbers of our sum. Note that we didn't have to assign the result of the [0-9]+ rule in the `num` rule. This is because rules that are just references to other rules are saved implicitly.

```
eoin@tsPEG$ tspeg grammar.peg parser.ts
eoin@tsPEG$ tsc
eoin@tsPEG$ node
> var P = require('./parser');
undefined
> new P.Parser('2+3').parse()
ParseResult { ast: { kind: 0, left: '2', right: '3' }, err: null }
>
```

Figure A.2: Sum example

Now that we have saved the numbers of our sum, it's easy to write a program to compute the result of adding the numbers. Each rule that assigns results to the AST is exported as a class or interface with the same name as the rule, so we can just import the `sum` interface from the parser to write our function. For this grammar we get an interface like:

```
interface sum {
    left: string;
    right: string;
}
```
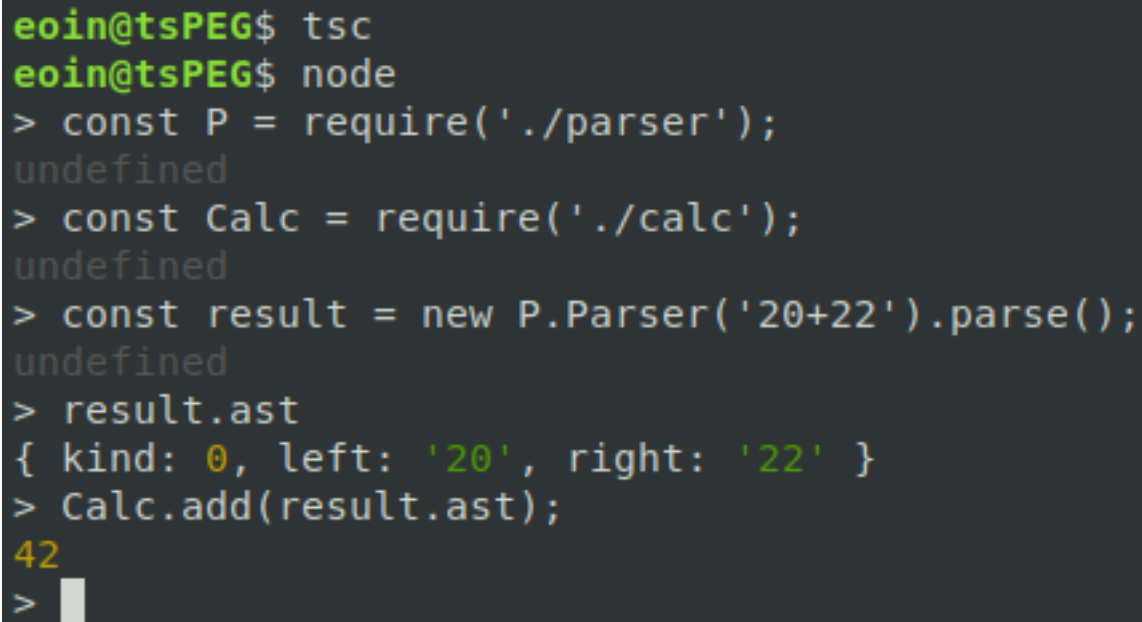
Allowing us to write our calculator function like:

```
import { sum } from "./parser";

export function add(ast: sum): number {
    return parseInt(ast.left) + parseInt(ast.right);
}
```

Calling this file "calc.ts" we can use it to calculate our sums:

We can also use computed properties to calculate the result of the sum during the parsing process.

```
eoin@tsPEG$ tsc
eoin@tsPEG$ node
> const P = require('./parser');
undefined
> const Calc = require('./calc');
undefined
> const result = new P.Parser('20+22').parse();
undefined
> result.ast
{ kind: 0, left: '20', right: '22' }
> Calc.add(result.ast);
42
>
```

Figure A.3: Calc screenshot

A.1.6   Operators

tsPEG supports a set of powerful operators to build complex grammars. The only operator we have seen so far is the | operator, which allows us to make choices between two rule expressions, however there are many more.

- The ? operator is used to make a match optional. For example in the rule

  ```
  rule := 'I ' 'really '? 'love tsPEG'
  ```

  The match for the 'really' string has been marked optional, so this rule can match either "I really love tsPEG", or "I love tsPEG".

- The + operator matches 1 or more copies of the match it's applied to, for example:

  ```
  rule := 'It\'s a ' 'long '+ 'way away'
  ```

  This rule matches "It's a long way away", "It's a long long way away", etc. for any amount of "long"s that's not 0. When this operator is used, a list of results is attached to the AST.

- The * operator is the same as the + operator but it allows zero matches.

- The ! is called the "*negative lookahead*" operator, and it does exactly what it says on the tin. This operator inverts the result of the match, meaning you can specify a rule by what it should not match. For example the rule `rule := 'The banned word is ' !'Macbeth'` This will match the phrase "The banned word is X" for any value of X, except when X is 'Macbeth'

- The & operator is called the "*postitive lookahead*" operator, this operator will change a match so that it will test for the match, and fail if it doesn't work, but it will not consume the input. This allows you to lookahead at what comes next in the string, but not to consume it.

A.1.7 Syntax Errors

A `SyntaxErr` object is composed of two fields, a `pos` field with the position of the error, and `expmatches` which contains a list of expected matches.

```
class SyntaxErr {
    pos: PosInfo;
    expmatches: string[];
}
class PosInfo {
    overallPos: number;
    line: number;
    offset: number;
}
```

A.1.8 Computed Properties

As well as assigning parsing results to variables and storing them on the AST, *tsPEG* also allows you to create **computed properties**, which are fields on the AST that are computed when the parser is run.

Computer properties are added to a rule by appending a new expression after the rule description like

```
.<propertyname> = <type> { <code to calculate property> }
```

Returning to our sum example from earlier we had a grammar to match strings like "2+3", "40+200" etc.

```
sum := left=num '\+' right=num
num := '[0-9]+'
```

We can add computed properties to this to compute the value of this sum at parse time, instead of writing our own function to do it after. First we add a computed property to `num` to store the value of the number:

```
sum := left=num '\+' right=num
num := literal='[0-9]+'
        .value = number { return parseInt(this.literal); }
```

As you can see we've assigned the text match to the field `literal`, and added a computed property called `value` which is the `literal` field parsed as an integer.

Now we can add a computed property to `sum` to do the arithmetic, this is very simple

```
sum := left=num '\+' right=num
        .sum = number { return this.left.value + this.right.value }
num := literal='[0-9]+'
        .value = number { return parseInt(this.literal); }
```

We use the computed property of the `num` rule to compute our `sum` property. Let's see this in action! Let's try and parse the string "240+100".

As you can see the AST has a field called `sum` with the correct value of 340 in it. The `left` and `right` also have their computed `value` properties of 240 and 100.

```
eoin@tsPEG$ tspeg grammar.peg parser.ts
eoin@tsPEG$ tsc
eoin@tsPEG$ node
> const P = require('./parser');
undefined
> new P.Parser('240+100').parse();
ParseResult {
  ast:
   sum {
      kind: 0,
      left: num { kind: 1, literal: '240', value: 240 },
      right: num { kind: 1, literal: '100', value: 100 },
      sum: 340 },
  err: null }
>
```

Figure A.4: Computed property

A.1.9   Header

The introduction of computed properties means that now you might want to import some types or functions into your parser. Luckily you can specify a header in the file that will be inserted directly into the generated parser. Anything at the top of the grammar file between two lines of three dashes `---` will be inserted straight into the generated parser, allowing you to write grammars like

```
---
import { myFunc, myType } from "./mypackage";
---
rule := hello='Hello World'
        .value = myType { return myFunc(this.hello); }
```