Patrick Dubroy / Mar 05 2023
with Martin Kavalar, Dieter Komendera, Joshua Sierles and Gregor Koehler

Ohm: Parsing Made Easy

Patrick Dubroy, co-author of Ohm

Ohm (https://ohmjs.org) is a parsing library for JavaScript, which was created at HARC (https://github.com/harc) to support our programming language research. We think of it as a *language implementation toolkit* that lets you quickly prototype new languages and experiment with extensions to existing languages. You can use Ohm to parse custom file formats or quickly build parsers, interpreters, and compilers for programming languages.

In this article, we'll introduce the basic features of Ohm by creating a simple arithmetic language and writing an interpreter for that language. When we're done, we'll have a desktop calculator that can evaluate expressions like `100 * 2 + 3`.

Setup

Browser

The quickest way to use Ohm in the browser is to load it directly from unpkg (https://unpkg.com), by adding the following script tag to your page:

```html
<script src="https://unpkg.com/ohm-js@17/dist/ohm.min.js"></script>
```
HTML

This introduces a new global variable named `ohm`.

Node.js

Under Node.js, you'll first need to install the `ohm-js` package using npm (https://www.npmjs.com/):

```bash
npm install ohm-js
```
Bash

...then use `require` to load the Ohm module into your script:

```javascript
const ohm = require('ohm-js');
```
JavaScript

Getting Started

Ohm consists of two parts: a domain-specific language, and a library. The *Ohm language* is based on parsing expression grammars (https://en.wikipedia.org/wiki/Parsing_expression_grammar) (PEGs), which are a formal way of describing syntax, similar to regular expressions and context-free grammars. The *Ohm library* provides a JavaScript interface for creating parsers, interpreters, and more from the grammars you write.

📝 *When writing grammars, we recommend using the Ohm Editor (https://ohmjs.org/editor) — its parsing visualization can be a huge aid to understanding and debugging.*

Creating a Grammar

The first step in defining a new language with Ohm is to create a grammar. Here is a very simple grammar for a language named "Arithmetic":

```
Arithmetic {
  Exp = "42"
}
```

Ohm

A grammar is made up of *rules*. This grammar has a single rule named "Exp" whose *rule body* is the literal string "42". To use this grammar, we must first instantiate a grammar object using the Ohm library:

```javascript
ohm.grammar(`
  Arithmetic {
    Exp = "42"
  }
`);
```

arithmetic0

Javascript ✔

📝 *In this article, our grammar definitions use ES6 template literal syntax (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Template_literals), because it's the only standardized way to do multi-line strings in JavaScript.*

Matching Input

We can use the grammar object's `match` method to recognize arithmetic expressions in our library. `match` returns a MatchResult object which (among other things) has a `succeeded` method:

```javascript
const matchResult = arithmetic0.match('42');
matchResult.succeeded()
```

Javascript ✔

Our arithmetic grammar doesn't actually *do* much yet — it successfully matches against the string "42", but anything else will fail:

```javascript
const matchResult = arithmetic0.match('1 + 2');
matchResult.succeeded()
```

Javascript ✔

📝 *For more, see the documentation for [Grammar.match() (https://github.com/harc/ohm/blob/master/doc/api-reference.md#Grammar.match)](https://github.com/harc/ohm/blob/master/doc/api-reference.md#Grammar.match).*

Parsers and Parse Trees

Though we called it a "grammar object", we could also say that `arithmetic0` is a *parser*. A parser is just a tool that takes some input and produces a structural representation of that input, typically in the form of a *parse tree*.

In Ohm, you do not directly deal with parse trees — though every successful MatchResult object contains a parse tree internally. However, seeing the parse tree for a given input makes it easier to understand how an Ohm grammar works. For that reason, as we build up the arithmetic grammar in this article, many of the examples will include a parse tree visualization taken from the [Ohm Editor (https://ohmjs.org/editor/)](https://ohmjs.org/editor/).

```
true
```

Recognizing Numbers

First, let's modify our grammar to recognize numbers (other than 42 that is). Here's one way we could do that:

```
Arithmetic {
  Exp = number
  number = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"
}
```

Ohm

The `|` operator represents a *choice* between different alternatives. This definition means that to match a number, Ohm first tries to match the character "0", then if that fails, it tries to match the character "1", etc.

Character Ranges

However, since it's fairly common to want to match a particular character range like this. Using the *range operator* (`..`), we can express the same thing in a clearer and more compact way:

```
  number = "0".."9"
```

Ohm

But in this case, we can just use Ohm's built-in "digit" rule, which does the same thing:

```
  number = digit
```

Ohm

Repetition Operators

To match numbers with more than one digit, we use the `+` operator, which means that the preceding expression should be matched one or more times:

```
ohm.grammar(`
  Arithmetic {
    Exp = number
    number = digit+
  }
`)
```

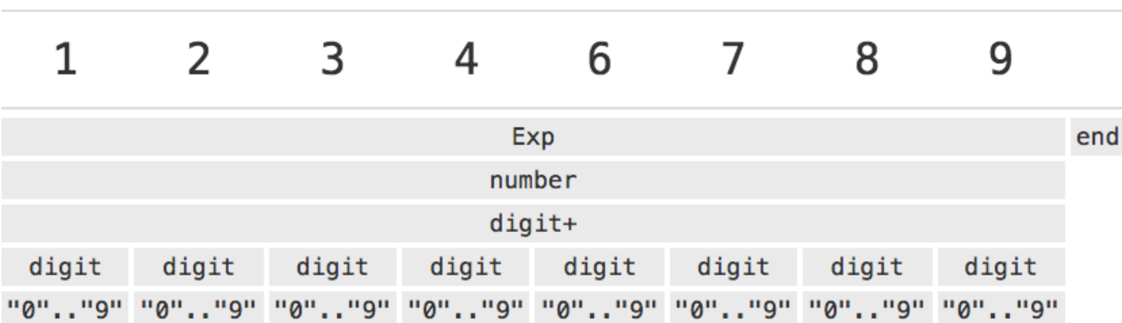
arithmetic1 Javascript

This version of the grammar can recognize numbers with any number of digits:

```
[
  arithmetic1.match('1').succeeded(),
  arithmetic1.match('99').succeeded(),
  arithmetic1.match('123456789').succeeded()
]
```

Javascript

Here is the parse tree for the last example:



Parse tree for the input "123456789"

Other Repetition Operators

The other repetition operator supported by Ohm is the *Kleene star* (`*`), which matches an expression *zero* or more times. So another way of defining the "number" rule would be `number = digit digit*`.

Addition and Subtraction

Now that we can recognize whole numbers, let's extend the grammar to support addition and subtraction:

```
ohm.grammar(`
  Arithmetic {
    Exp = Exp "+" number  -- plus
        | Exp "-" number  -- minus
        | number
    number = digit+
  }
```

```
`)
```

arithmetic2

Javascript

We've changed the body of the "Exp" rule to be a choice (or *alternation*) with three branches: one for addition, one for subtraction, and a final branch that just matches a number.

The text `-- plus` and `-- minus` are known as *case labels*. They do not affect *what* input is matched, but they do affect *how* that input is matched. A full explanation will come later, but for now, you can think of them as comments that document the purpose of a branch.

Left Recursion

▶Array(3)[true, true, true]

Another interesting thing to note about the new definition of "Exp" above is that it is recursive — i.e., its body contains an application of "Exp" itself. More specifically, it is *left recursive (https://en.wikipedia.org/wiki/Left_recursion)*, meaning that the recursive application is the first expression in the branch.

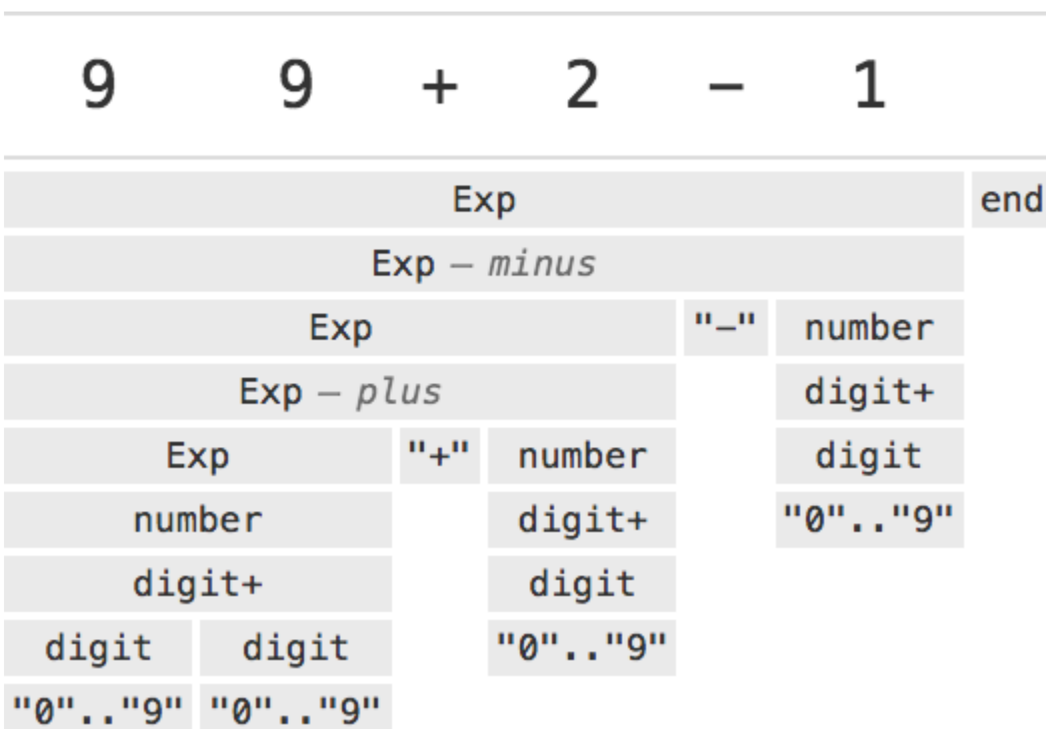
Many PEG-based parser generators do not support left recursion — requiring grammar authors to use repetition or right recursion instead. But left recursion is the most straightforward way to express left associative operators, which is why left recursion is supported by Ohm.

Before we move on, let's verify that our grammar can successfully recognize addition and subtraction:

```
[
arithmetic2.match('99').succeeded(),arithmetic2.match('99 + 1 - 2').succeeded()
]
```

Javascript

Below is the parse tree for "99 + 2 - 1":



The parse tree for "99 + 1 - 2"

Note that the case labels ("— minus" and "— plus") actually appear in the tree, indicating which branch succeeded in matching the input.

Multiplication and Division

▶ {name: "Arithmetic", superGrammar: Grammar, rules: Object, defaultStartRule: "Exp"}

Adding support for multiplication and division can be done in a similar way — we just need to add two more cases to the "Exp" rule:

```
ohm.grammar(`
  Arithmetic {
    Exp = Exp "+" number  -- plus
        | Exp "-" number  -- minus
        | Exp "*" number  -- times
        | Exp "/" number  -- div
        | number

    number = digit+
  }
`)
```

arithmetic3 Javascript ✔

And once again, let's verify that it works as expected:

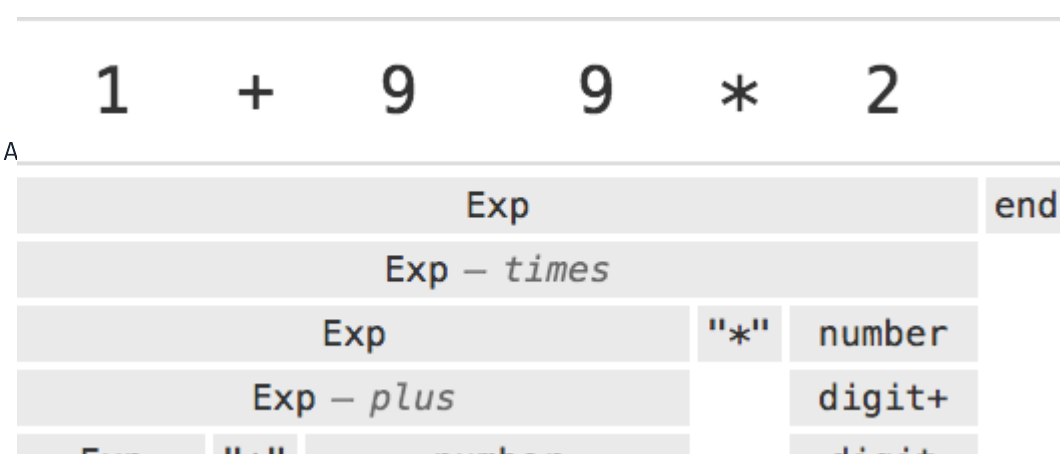
```
[
  arithmetic3.match('1 + 99 * 2').succeeded(),
  arithmetic3.match('1024 / 256').succeeded(),
]
```

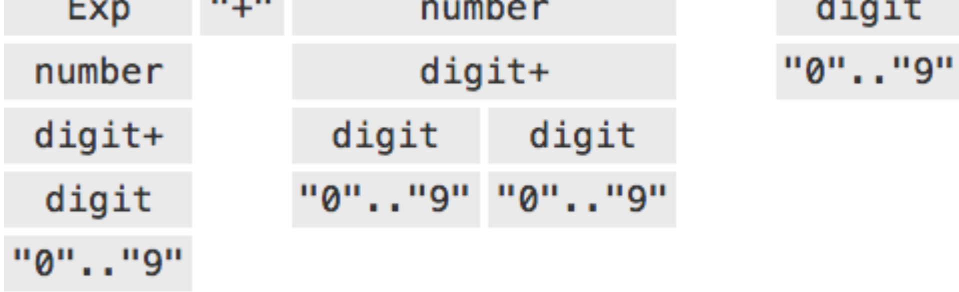
Javascript ✔

Operator Precedence

Our grammar can now parse arithmetic expressions, but it still has one problem: all of the operators are treated equally. This can be seen in the parse tree for "1 + 99 * 2":

▶ A





Parse tree for "1 + 99 * 2" with no operator precedence

Notice that "plus" case appears lower in the tree than the "times" case. This is fine when we are just *recognizing* expressions, but when it comes time to evaluate them, this structure will make things difficult.

In Ohm, the simplest way to handle precedence is to encode it in the grammar. To do this, we should first refactor the "times" and "div" cases into a separate "MulExp" rule:

```
MulExp = MulExp "*" number  -- times
       | MulExp "/" number  -- div
       | number
```

Ohm

Then, we'll do the same with the "plus" and "minus" cases, but with one small change — replacing applications of "number" with "MulExp":

```
AddExp = AddExp "+" MulExp  -- plus
       | AddExp "-" MulExp  -- minus
       | MulExp
```

Ohm

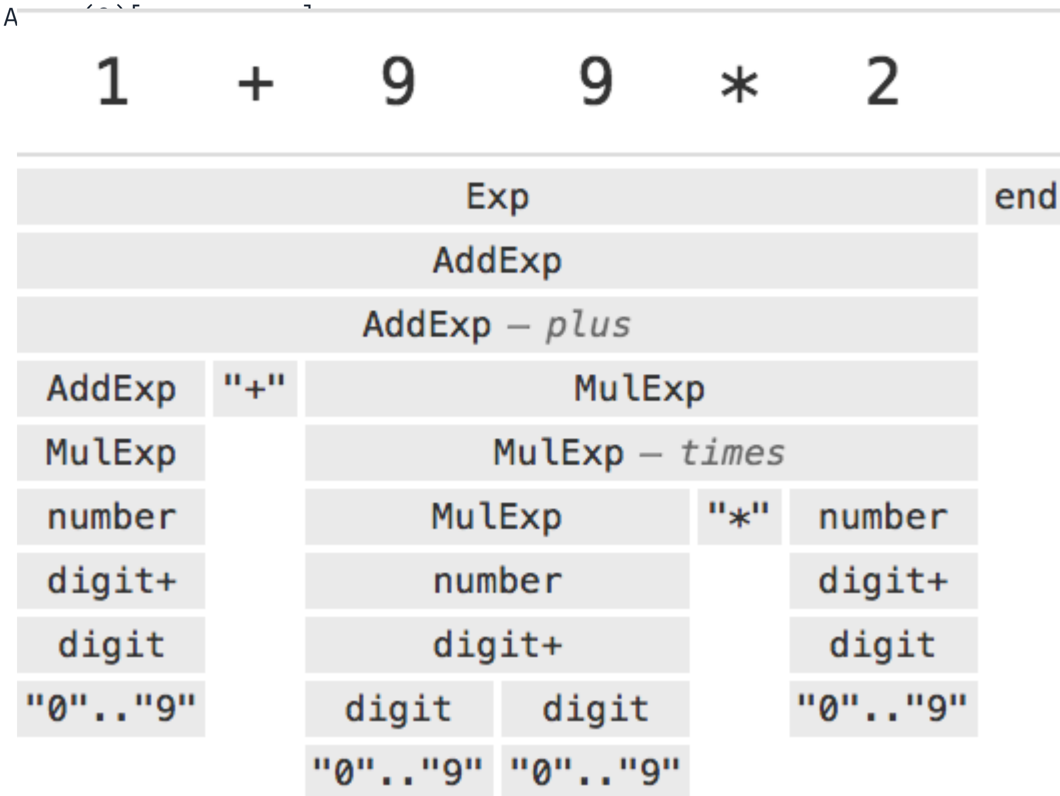
Here is what the grammar looks like after refactoring:

```
ohm.grammar(`
  Arithmetic {
    Exp = AddExp

    AddExp = AddExp "+" MulExp  -- plus
           | AddExp "-" MulExp  -- minus
           | MulExp

    MulExp = MulExp "*" number  -- times
           | MulExp "/" number  -- div
           | number

    number = digit+
  }
`)
```

arithmetic

Javascript ✓

And here is the new parse tree for "1 + 99 * 2":



Parse tree for "1 + 99 * 2" with correct operator precedence

Notice that the "times" case now appears lower in the tree, indicating that the `*` operator binds more tightly than `+`, as it should.

For the purposes of this tutorial, our arithmetic grammar is done. In the next section, we'll move on to evaluating arithmetic expressions. Alternatively, you can open it in the Ohm Editor (https://ohmlang.github.io/editor/#df0722395daf6df4cd99e02188b9c030) and keep experimenting.

Defining Semantics

While the grammar above defines the *syntax* of the arithmetic language, it doesn't specify the *semantics* — i.e., what to do with valid inputs.

In many parser generators (e.g. Yacc and ANTLR), a grammar author can specify the language semantics by including *semantic actions* inside the grammar. A semantic action is a snippet of code — typically written in a different language —that produces a desired value or effect each time a particular rule is matched.

But Ohm is a bit different: it completely separates grammars from semantic actions. In Ohm, a grammar defines a language, and semantic actions are written separately using the Ohm library. One advantage of this approach is that a single grammar can have more than one semantics associated with it.

Action Dictionaries

In Ohm, a set of semantic actions for a grammar is usually written using an object literal. E.g.:

```
const actions = {
  Exp() { ... },
  AddExp() { ... },
```

```
  MulExp() { ... },
  number() { ... }
};
```

JavaScript

Each key in the object is a rule name, and the value is the rule's semantic action (a function). We call this kind of object an *action dictionary*. There are three specially-named keys that can also appear in an action dictionary:

- "_terminal" specifies the action to use for all *terminal expressions* (e.g. , `"abc"`).
- "_nonterminal" species a catch-all action for all other expresssions. This is analogous to Smalltalk's `doesNotUnderstand:` or Ruby's `method_missing`.
- "_iter" is used for repetition expressions.

Before we talk about how to write the various semantic actions, let's first discuss how semantic actions are used in Ohm.

Defining an Operation

To associate a set of semantic actions with a particular grammar, we first need to create a *Semantics object* for that grammar, using its `createSemantics` method:

```
const s = arithmetic.createSemantics();
```

JavaScript

Then, we add a new *operation* to the semantics, by calling its `addOperation` method with an action dictionary as the argument:

```
s.addOperation('myOperation', {
  Exp() { ... },
  AddExp() { ... },
  MulExp() { ... },
  number() { ... }
});
```

JavaScript

Using an Operation

In the next section, we'll take a closer look at the details of the semantic actions, but first, let's talk about how to use an operation after you've defined it. If you look at the result of createSemantics, you'll see that it actually returns a function:

```
typeof arithmetic.createSemantics();
```

Javascript ✔

This function takes a single argument: a `MatchResult` object, as returned by the grammar's `match` method:

```JavaScript
const matchResult = arithmetic.match('100 * 2 + 3');
const adapter = s(matchResult);
```

The result of invoking the semantics function is an object called a *semantic adapter*. A semantic adapter is just an object that gives you a way to execute operations on a particular `MatchResult`. If you defined an operation called "myOperation", the adapter will have a `myOperation` method that you can call:

```JavaScript
adapter.myOperation();
```

Now that we've explained how to define and create operations, let's take a look at how to write semantic actions.

Writing Semantic Actions

In the previous section, we introduced Ohm's concept of an *operation*, which is defined by a name and a set of semantic actions. The semantic actions are stored in an *action dictionary*, which provides a mapping between a rule name and its semantic action (a JavaScript function). In this section, we'll write the semantic actions for an operation called `eval` that will evaluate expressions in the arithmetic grammar.

Let's begin with an action for the `number` rule:

```JavaScript
  number(_) {
    return parseInt(this.sourceString);
  }
```

This action takes the text that is matched by the "number" rule — e.g., "100" — and converts it to an actual number using JavaScript's built-in `parseInt` function. Inside any semantic action, you can always use `this.sourceString` to get the raw text matched by that node.

Note that this action takes a single argument, but its value is ignored. (In JavaScript, the underscore character is a valid identifier; naming an argument `_` is just a convention to indicate that its value is not used.)

Rule Arity

The number of arguments a semantic action takes is determined by the *arity* of the body of its corresponding rule. In general, the arity of an expression is equal to the "number of things" matched by that expression. For example, recall the definition of the "Exp" rule:

```Ohm
  Exp = AddExp
```

The semantic action for "Exp" will have one argument, because the rule body consists of a single expression: an application of the "AddExp" rule.

Similarly, the body of "number" consists of a single expression:

Similarly, the body of "number" consists of a single expression:

```
number = digit+
```

Ohm

...so the semantic action for "number" also takes a single argument.

Argument Types

Each argument to a semantic action is a semantic adapter, just like the ones that are used to execute an operation. Hopefully this helps make their purpose more clear: **a semantic adapter is an interface to a particular parse tree node**, providing a way to execute operations on that node.

To invoke an operation on an adapter, you just call the appropriate method, e.g., `eval`. For example, here is a possible semantic action for the "Exp" rule:

```
Exp(e) {
  return e.eval();
}
```

JavaScript

The argument `e` is a semantic adapter for an "AddExp" node. The meaning of this action is: the result of evaluating an "Exp" node is the result of evaluating its only child. We call this a *pass-through action*, and because it's such a common case, you can actually omit these actions entirely. **If you don't specify a semantic action, Ohm will use a pass-through action by default** (as long as the action only takes a single argument).

Case Labels and Arity

To complete the discussion of arities, let's look at the definition of "AddExp":

```
AddExp = AddExp "+" MulExp  -- plus
       | AddExp "-" MulExp  -- minus
       | MulExp
```

The first and second branches — labeled "plus" and "minus", respectively — each have three subexpressions. The last (unlabeled) branch only has one. So how many arguments should the semantic action for "AddExp" take?

Suppose it took three arguments — then, inside the action, you might check `arguments.length` to determine which case succeeded. This would work, but it's awkward an error-prone. For these reasons, **Ohm requires every branch of an expression to have the same arity.**

One way to eliminate the amibiguity would be to refactor the "plus" and "minus" cases into their own rules:

```
AddExp = AddExp_plus
       | AddExp_minus
       | MulExp
```

```
AddExp_plus = AddExp "+" MulExp

AddExp_minus = AddExp "-" MulExp
```

Now, each branch in "AddExp" has arity 1, and the "plus" and "minus" cases can be handled in separate semantic actions. The downside of this refactoring is that it makes the grammar more verbose.

Recall the case labels (e.g., "plus", "minus") that were introduced earlier, and how they actually appeared in the parse tree. In turns out **the case labels are actually a shorthand for declaring separate rules** named "AddExp_plus" and "AddExp_minus". In other words, the original declaration of "AddExp" is actually equivalent to the refactored version above.

This ensures that each branch in "AddExp" has the same arity, and it also makes it easy to write semantic actions for the three different cases — you simply write separate actions for "AddExp_plus" and "AddExp_minus":

```
AddExp_plus(a, _, b) {
      return a.eval() + b.eval();
},
AddExp_minus(a, _, b) {
  return a.eval() - b.eval();
},
```

These are pretty straightforward: evaluate the the two operands, and return the result of adding (or subtracting) them.

Note that in both actions, the second argument is ignored. It represents the operator, but because of the case labels, we don't need to examine it.

The actions for "MulExp_times" and "MulExp_div" follow the same pattern:

```
MulExp_times(a, _, b) {
  return a.eval() * b.eval();
},
MulExp_div(a, _, b) {
  return a.eval() / b.eval();
},
```

Putting It All Together

Here is a complete definition of the "eval" operation, with the unnecessary pass-through actions omitted:

```
const semantics = _.createSemantics();
semantics.addOperation('eval', {
```

```
semantics.addOperation('eval', {
  AddExp_plus(a, _, b) {
        return a.eval() + b.eval();
  },
  AddExp_minus(a, _, b) {
    return a.eval() - b.eval();
  },
  MulExp_times(a, _, b) {
    return a.eval() * b.eval();
  },
  MulExp_div(a, _, b) {
    return a.eval() / b.eval();
  },
  number(digits) {
    return parseInt(digits.sourceString)
  }
});
```

We can use this operation to evaluate arithmetic expressions:

```
[
  _(_.match('100 + 1 * 2')).eval() == 102,
  _(_.match('1 + 2 - 3 + 4')).eval() == 4,
  _(_.match('12345')).eval() == 12345
]
```

Multiple Operations

Ohm's approach to semantic actions clearly requires more work than just embedding the actions directly in the grammar — so why do we do things this way?

If semantic actions are embedded in the grammar, the grammar assumes a particular interpretation. But often, it makes sense to use the same grammar to process the input in multiple ways. For example, you might want to support evaluation, syntax highlighting, and pretty printing — all from the same grammar.

One way to do this in Ohm is to add multiple operations to the same semantics. E.g., to add a "prettyPrint" operation, we can just use the `addOperation` method again:

```
semantics.addOperation('prettyPrint', { /* ... */ })
```

Now, any adapter objects we create using `semantics` will have both a `prettyPrint` method and an `eval` method:

```
const adapter = semantics(g.match('100 * 2 + 3'));
const result = adapter.eval();
const prettyExp = adapter.prettyPrint();
```

This is how Ohm's notion of *operations* makes it possible to use different sets of semantic actions with the same grammar. In fact, the operations in a semantics can even depend on each other — that's why we say that **a semantics is a family of operations**.

Further Reading

Hopefully this article has given you a good taste of what it's like to work with Ohm. To learn more, take a look at some of the following pages:

- Documentation index (https://ohmjs.org/docs/intro)
- Syntax reference (https://ohmjs.org/docs/syntax-reference)
- API reference (https://ohmjs.org/docs/api-reference)

You might also find it useful to use the Ohm Editor to experiment with real-world grammars, such as a more fully-featured arithmetic language (https://ohmjs.org/editor/#30325d346a6e803cc35344ca218d8636), or a complete ES5 grammar (https://ohmjs.org/editor/#0a9a649c3c630fd0a470ba6cb75393fe).

If you have questions, you can join us on Discord (https://discord.gg/KwxY5gegRQ), GitHub Discussions (https://github.com/harc/ohm/discussions), or the Ohm mailing list (https://groups.google.com/a/ycr.org/forum/#!forum/ohm) — there are plenty for friendly folks who are happy to help.

Happy parsing!✌️

▼ Appendix

```
require('https://unpkg.com/ohm-js@17/dist/ohm.min.js')
```

Runtimes (1)