Modular Semantic Actions

Alessandro Warth Patrick Dubroy
Y Combinator Research, USA
{alex.warth,pat.dubroy}@ycr.org

Tony Garnock-Jones
Northeastern University, Boston, USA
tonyg@ccs.neu.edu

Abstract

Parser generators give programmers a convenient and declarative way to write parsers and other language-processing applications, but their mechanisms for extension and code reuse often leave something to be desired. We introduce Ohm, a parser generator in which both grammars and their interpretations can be extended in safe and modular ways. Unlike many similar tools, Ohm completely separates grammars and semantic actions, avoiding the problems that arise when these two concerns are mixed. This paper describes the particular way in which Ohm achieves this separation, and discusses the resulting benefits to modularity and extensibility.

Categories and Subject Descriptors D.1.5 [*Programming Techniques*]: Object-Oriented Programming; D.3.3 [*Programming Languages*]: Language Constructs and Features

Keywords parser generators, modularity, extensibility

1. Introduction

Parser generators and compiler-compilers like Yacc [9], ANTLR [14], and OMeta [20] give programmers a convenient and declarative way to write parsers and other language-processing applications. These tools are built around the notion of a *grammar*, which the programmer writes to specify the syntax of the language.

As an example, the following is a fragment of an OMeta/JS [19] grammar for arithmetic expressions:

```
mulExp = mulExp "*" priExp
       | priExp

priExp = "(" exp ")"
       | number
```

A "pure" grammar like the one above does not specify what to do with valid inputs. To do this, the programmer writes *semantic actions* in the grammar. A semantic action is a snippet of code — typically written in a different language — that, when executed, will produce the desired value or effect. Together, the semantic actions give a specific meaning (an interpretation) to the language defined by the grammar.

Here is how a programmer might add semantic actions to the grammar fragment above in order to implement a calculator:

```
mulExp = mulExp:x "*" priExp:y -> (x * y)
       | priExp:e              -> e

priExp = "(" exp:e ")"         -> e
       | number:n              -> n
```

The first semantic action in the `mulExp` rule, for instance, computes the value of a "times" expression. We give names to two sub-expressions (`x` and `y`) so that we can refer to their values in the semantic action, which is written to the right of the `->`. The values of the sub-expressions themselves are also computed by semantic actions. Note that *bindings* — assignments of names to sub-expressions — and semantic actions are not specific to OMeta; most parser generators (e.g., Yacc and ANTLR) work this way, though their syntax for bindings and semantic actions may differ.

Mixing grammars and semantic actions in this way is convenient for small examples, but it sacrifices the following important properties:

- **Modularity.** The second version of our arithmetic grammar is no longer a grammar but an interpreter. And because the grammar and semantic actions are mixed, if we later realize that we also need a compiler or syntax highlighter for the same language, we will have to duplicate the part of the code that specifies the syntax of the language. From this point on, we will have to manually ensure that the language that is accepted by these different grammars is the same, which is both difficult and error-prone.
- **Readability.** Grammars are cluttered with bindings and semantic actions, which makes it more difficult for programmers to "see" the syntax of the language that they specify.
- **Extensibility.** Parser generators that support grammar inheritance, e.g., OMeta, ANTLR, and *Rats!* [8], enable pro-

DLS'16 November 1, 2016, Amsterdam, Netherlands
DOI: http://dx.doi.org/10.1145/2989225.2989231

grammers to override rules from the parent grammar, do "super-sends", etc. But while these mechanisms provide adequate support for extending the syntax of a grammar, they do not make the task of providing different semantic actions any easier: as discussed above, to override a semantic action (e.g., to change the implementation of "times" expressions in our calculator) the programmer has to duplicate the code that specifies the syntax that is associated with it.

In this paper, we introduce Ohm, a new parser generator based on Parsing Expression Grammars (PEGs) [5] that enables programmers to write modular, readable, and extensible grammars as well as semantic actions. Our design accomplishes this by enforcing a strict separation between grammars and semantic actions, both *in code*, and *in time*:

- **In code.** The fact that Ohm's grammars and semantic actions are written separately — as in Newspeak's Executable Grammars [1] — leads to more readable grammars, and makes it possible for grammars and semantic actions to be extended independently, using familiar object-oriented mechanisms. (More on this in Section 4.)
- **In time.** In Ohm, it is the grammar's job (and only the grammar's job) to determine whether an input is valid; semantic actions are only evaluated on valid inputs.[1] This leads to a simpler programming model, as the programmer does not have to worry about the interaction of semantic actions with backtracking.

The contributions of this paper are (i) the design of the Ohm parser generator, and (ii) the following novel concepts, which not only support the separation of grammars from semantic actions, but also support each other:

- Static checks (Section 2.5) ensure that semantic actions are "compatible" with the grammar. This can prevent later changes to the grammar from introducing subtle bugs to existing semantic actions.
- A novel distinction between *lexical* and *syntactic* rules (Section 3) gives programmers a more readable and less error-prone way to handle whitespace in their grammars. (Say goodbye to those pesky "`space*`" expressions!) This feature also makes writing semantic actions more convenient in the presence of the static checks.
- Ohm's notion of a *semantics* (Section 2.2), which is a collection of *operations* that can be invoked on valid inputs.
- Extensibility mechanisms for grammars and semantics (Section 4) that "play nicely" with the aforementioned static checks. Ohm's grammars and semantics can be extended independently of each other, e.g., a programmer can create a new semantics by "subclassing" an existing semantics without also extending its associated grammar.

[1] This means that Ohm does not support *semantic predicates*, which results in a small loss of expressiveness, e.g., it is not currently possible to write Ohm grammars for context-sensitive languages like Python and HTML. We believe this is a fair price to pay for the increased modularity, readability, and extensibility that our system provides.

```
Arithmetic {
  exp = addExp

  addExp = addExp "+" mulExp  -- plus
         | addExp "-" mulExp  -- minus
         | mulExp

  mulExp = mulExp "*" priExp  -- times
         | mulExp "/" priExp  -- div
         | priExp

  priExp = "(" exp ")"  -- paren
         | num

  num = dig+

  dig = "0".."9"
}
```

Figure 1. Ohm grammar for a language of arithmetic expressions.

The remainder of this paper is structured as follows. Section 2 introduces Ohm's grammars, semantics, and static checks by example. Section 3 builds on that example to explain and demonstrate the benefits of syntactic rules. Section 4 describes and illustrates Ohm's support for extensibility and how it interacts with the static checks. Section 5 outlines our experience with Ohm. Section 6 discusses related work, and Section 7 concludes.

2. Ohm by Example

Ohm consists of (i) a PEG-based language for writing grammars, and (ii) a library that enables programmers to match inputs against a given grammar as well as define and use semantics for their grammars. At the time of this writing, there are two ohm implementations: Ohm/JS [21], which is our JavaScript-based reference implementation, and Ohm/S [15], for Smalltalk. Ohm grammars are platform-independent: there is only one syntax for writing them, and the same grammar can be used in either of the Ohm implementations. The libraries, on the other hand, are platform-specific. All of the examples in this paper are written in Ohm/JS.

In this section, we explain the rationale and implications of Ohm's design in the context of a running example: a language of arithmetic expressions. The full grammar for this language is shown in Figure 1.

An Ohm grammar has a name (e.g., `Arithmetic`) and contains a set of *rules*. Each rule has a name (e.g., `exp`, `addExp`, etc.) and a *body*, which can be any valid *parsing*

Expression	Meaning
`"abc"`	terminal (matches the characters `abc`)
`"a".."z"`	character range
$e_1\ e_2$	sequencing
$e_1 \mid e_2$	prioritized choice
x	rule application
x<e_1, ..., e_n>	parameterized rule application
e*	zero or more repetitions
e+	one or more repetitions
&e	lookahead
$\sim e$	negation
#e	lexification

Table 1. The language of Ohm parsing expressions.

expression. The language of parsing expressions supported by Ohm is summarized in Table 1.

2.1 Recognizing Input

In order to use this grammar to recognize inputs, we must instantiate it using the Ohm API. If the grammar source is stored in a file named "arithmetic.ohm", we can do this as follows:

```
var g =
    ohm.grammarFromFile('arithmetic.ohm');
```

Now we can use the grammar object's `match` method to recognize inputs. `match` returns a `MatchResult` instance, which (among other things) can be used to check whether the match succeeded or failed. For example:

```
g.match('1+2*3').succeeded();  // true
g.match('blerg').succeeded();  // false
```

The `match` method takes an optional second argument that specifies the name of the start rule. If the start rule is not specified, it uses the grammar's *default start rule*, which is the first rule defined in the grammar — in this case, `exp`.

2.2 Specifying and Using Semantics

Like context free grammars, PEGs describe the syntax of a language, but they do not specify what to do with valid inputs in that language — that is the responsibility of semantic actions. Unlike most parser generators, Ohm does not allow semantic actions to be written directly in a grammar. Instead, semantic actions are written in the host language using the Ohm library. In Ohm/JS, a hypothetical semantic action for the arithmetic grammar's `num` rule might look like this:

```
var numAction = function(ds) {
  return parseInt(ds.sourceString);
};
```

This function produces a semantic value for an input string that matches the grammar's `num` rule. For example, for an input string containing the text `42` (as a string), it would return the number `42`. It does this by calling JavaScript's built-in `parseInt` function, passing it the portion of the input stream that was consumed by `num`'s body expression (which is available as `ds.sourceString`).

Typically, the programmer writes a *set* of semantic actions, one for each of the rules in the grammar. This set forms a family of related actions, much like different methods in a generic function. We call a family of semantic actions an *operation*. In Ohm/JS, when the programmer creates an operation, he must provide an object that maps rule names to semantic actions.

Consider the semantic actions for an operation that evaluates expressions in our arithmetic grammar[2]:

```
var numActions = {
  exp(e) { ... },
  addExp(e) { ... },
  mulExp(n) { ... },
  num(ds) {
    return parseInt(ds.sourceString);
  },
  ...
}
```

This is known as an *action dictionary*, and it is simply a JavaScript object whose properties match up with the names of rules in the associated grammar. The value of each property is a function that implements the semantic action for its corresponding rule.

There are three "special" action names that can also appear in action dictionaries:

- `_nonterminal` is a catch-all action, similar to Smalltalk's `doesNotUnderstand:` or Ruby's `method_missing`.
- `_terminal` is used for terminal expressions, e.g. `"abc"`.
- `_iter` is relevant to repetition expressions, which are discussed later in this paper.

These special action names offer a form of metaprogramming, making it easy for many different expressions to share the same semantic action —- without any boilerplate or duplicated code. In many parser generators (e.g., OMeta), we must write a separate semantic action for every rule in the grammar, even if they are nearly identical.

It is often convenient for one operation to depend on another. For example, a `prettyPrint` operation for arithmetic expressions might depend on a `toAST` operation. An operation is a family of actions, and **we call a family of operations a *semantics*.** Every operation in Ohm belongs to a particular semantics. To create a semantics for a given grammar, we call the grammar's `createSemantics()` method. Then, to add an operation to that semantics, we call its `addOperation` method, passing an action dictionary:

[2] The examples in this paper use some features of the ECMAScript® 2015 (also known as *ES6*) syntax — specifically, *method definitions* and *arrow function definitions*.

```
var s = g.createSemantics();
s.addOperation('eval()', {
  exp(e) { ... },
  ...
});
```

Invoking an operation. A semantics behaves like a function that takes a single (successful) MatchResult argument and returns a *semantic adapter*. A semantic adapter provides a method corresponding to each operation in the semantics. For example, we can invoke the *eval* operation by calling an adapter's `eval()` method:

```
s(g.match('1+2*3')).eval();  // Returns 7
```

Defining multiple operations. Since a semantics represents a *family* of operations, we can of course define other operations in the same semantics:

```
s.addOperation('toAST()', {
  exp(e) { ... },
  addExp(e) { ... },
  ...
});
```

Now the adapters created by this semantics will have both an `eval` and a `toAST` method. We can invoke `toAST` in the expected manner: `s(matchResult).toAST()`.

Defining multiple semantics. The same grammar may have more than one semantics associated with it. For example, the semantics declared below provides its own `typecheck` and `eval` operations:

```
var otherS = g.createSemantics();
otherS.addOperation('typecheck()', ...);
otherS.addOperation('eval()', ...);
```

The operations in `s` are completely separate from those in `otherS`, even when they have the same name (e.g., `eval`). This means that a grammar can be used in different ways, *even in the same program*, in a completely modular way. There is no risk of name clashes, accidental overriding, etc. Note that the same `MatchResult` instance can be used with more than one semantics, i.e., the programmer can access the functionality provided by different semantics for the same input, *without having to parse it multiple times*.

Now that we have seen how semantic actions are organized in the Ohm/JS API, we turn our attention to the mechanics of writing the semantic actions themselves.

2.3 Writing Semantic Actions

The number of arguments of a semantic action is determined by the *arity* of the body of its corresponding rule in the grammar. Informally, we can say that the arity of an expression is generally equal to the "number of things" matched by that expression. For example, recall the definition of the `exp` rule from Figure 1:

Expression	Arity
`"abc"`	1
`"a".."z"`	1
$e_1\ e_2$	$\text{Arity}(e_1) + \text{Arity}(e_2)$
$e_1 \mid e_2$	$\text{Arity}(e_1)$ *if* $\text{Arity}(e_1) = \text{Arity}(e_2)$ [3]
x	1
x<e_1, ..., e_n>	1
e*	$\text{Arity}(e)$
e+	$\text{Arity}(e)$
& e	$\text{Arity}(e)$
$\sim e$	0
# e	$\text{Arity}(e)$

Table 2. The arities of Ohm parsing expressions.

```
exp = addExp
```

A 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 `dig` rule:

```
dig = "0".."9"
```

also has an arity of 1. Its body is a *character range* expression, which consumes a single character from the input stream (in this case, any character from `0` to `9`, inclusive).

The arity of each type of parsing expression in Ohm is summarized in Table 2.

Each argument to a semantic action is a semantic adapter, just like the object produced by `s(matchResult)`. At this point, the role of adapters may be more clear: **a semantic adapter is an interface to a concrete syntax tree (CST) node.** It provides a way of evaluating semantic operations on a particular node. In fact, adapters are the *only* interface to CST nodes. This was a key design decision behind Ohm's modular semantics, which we discuss in detail in Section 5.

To invoke an operation on a semantics adapter, we simply call one of its methods, e.g., `eval()`. For example, here is a possible semantic action for the `exp` rule:

```
s.addOperation('eval()', {
  exp(e) {
    return e.eval();
  },
  ...
});
```

The call `e.eval()` returns the semantic value of `exp`'s body expression, which is provided by the `addExp` action — so we'll need to add that action too. Recall the definition of `addExp`:

```
addExp = addExp "+" mulExp  -- plus
       | addExp "-" mulExp  -- minus
       | mulExp
```

[3] An alternation containing expressions with different arities is a static error.

The body of this rule is an *alternation* of three parsing expressions. The expression in the first branch:

```
addExp "+" mulExp
```

has arity 3. That is, it will produce 3 CST nodes if it successfully matches the input: one for the `addExp` application, one for the terminal `"+"`, and one for the `mulExp` application. Likewise, the expression in the second branch also has arity 3. However, the third branch:

```
mulExp
```

only has arity 1.

This mismatch of arities in the three branches would be problematic for someone who is trying to write a semantic action for `addExp`. How many arguments should that semantic action take? It would be awkward if it depended on which branch succeeded — the programmer would be forced to check the value of `arguments.length` in order to implement an appropriate action. To make matters worse, that would make it impossible for Ohm to check the arities of semantic actions at operation creation time, which would in turn make programming with Ohm more error-prone. For these reasons, **Ohm requires every branch of an alternation expression to have the same arity.** This check is performed statically on the grammar.

We could address this by refactoring the first two choices of `addExp` 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 both `addExp_plus` and `addExp_minus` have arity 3, and everything is consistent. The downside of this refactoring is that it has made our grammar more verbose.

As a convenience, Ohm provides a syntactic sugar for this common construction: the programmer can write *case labels* (e.g., `-- plus`) as a way of declaring a new rule (e.g., `addExp_plus`) inline. Thus, the original arithmetic grammar shown in Figure 1 (which includes several case labels) desugars to the refactored version above.

Case labels are only required when the branches of an alternation have different arities. A simple alternation like `username | email` does not require case labels, since both branches have the same arity.

The case labels mean that to implement the semantics for `addExp`, we write three separate semantic actions for the main `addExp` rule:

```
addExp_plus(a, op, b) {
  return a.eval() + b.eval();
},
addExp_minus(a, op, b) {
  return a.eval() - b.eval();
},
addExp(e) {
  return e.eval();
}
```

Notice that the semantic action for the `addExp` rule is simply calling `eval()` on its only argument. In Ohm, we refer to this a *pass-through action*. Pass-through actions are very common, so as a convenience, any rule with arity 1 that does not have a semantic action automatically gets pass-through behavior. This prevents the programmer from having to write boilerplate code, and enables him to focus on the more interesting semantic actions.

The semantic actions for `mulExp` can be written in a very similar manner to `addExp`. We continue by writing a semantic action for the `priExp_paren` case, which is straightforward:

```
priExp_paren(open, e, close) {
  return e.eval();
},
```

The `num` rule is a bit more interesting. Recall its definition in the arithmetic grammar:

```
num = dig+
```

The rule body uses the one-or-more operator (`+`), which is one of the PEG *repetition operators*. No matter how many times `dig` is matched, the expression `dig+` is represented by a single *repetition node* in the parse tree — so the `num` action takes a single argument.

The semantic adapter for a repetition node has special behavior. Its interface is the same as any other adapter in the same semantics, but its methods return arrays, which are the result of mapping the operation over all of the matches. As an example, consider the following semantic actions for the `num` and `dig` rules:

```
num(ds) {
  var digits = ds.eval();
  return digits.reduce(
      (acc, d) => acc * 10 + d,
      0);
},
dig(d) {
  // In a semantic action, `this` is bound to
  // a semantic adapter for the current node.
  var ch = this.sourceString;
  return ch.charCodeAt(0) -
         "0".charCodeAt(0);
}
```

In `num`'s semantic action, the call to `ds.eval()` returns an array of one or more values, i.e., the results of invoking the `dig` action for each successful match of the `dig` rule.

However, we could write these actions more simply. The `num` action could simply use JavaScript's built-in `parseInt`

```
s.addOperation('eval()', {
  addExp_plus(a, op, b) {
    return a.eval() + b.eval();
  },
  addExp_minus(a, op, b) {
    return a.eval() - b.eval();
  },
  mulExp_times(a, op, b) {
    return a.eval() * b.eval();
  },
  mulExp_div(a, op, b) {
    return a.eval() / b.eval();
  },
  priExp_paren(open, e, close) {
    return e.eval();
  },
  num(ds) {
    return parseInt(ds.sourceString);
  }
});
```

Figure 2. Definition of an eval() operation for the arithmetic grammar.

to convert its entire matched input to a number. If we do that, then we can avoid writing a semantic action for `dig` altogether, because it is never used.

2.4 Putting It All Together

A complete definition of the `eval` operation is shown in Figure 2. It contains the simplified semantic action for `num` that we mentioned above, and omits the unnecessary pass-through actions for `eval`, `addExp`, `mulExp`, and `priExp`.

Together, the grammar defined in Figure 1 and the `eval` operation defined in Figure 2 implement a working calculator:

```
var matchResult = g.match('1+2*3');
s(matchResult).eval();  // Returns 7
```

2.5 Static Checks

While the separation of syntax and semantics has benefits for modularity, it also introduces some challenges. In particular, if the grammar is modified, it can easily fall out of step with the semantics. In order to mitigate these problems, `addOperation` performs two important static checks:

1. It verifies that each key in the action dictionary corresponds to a rule in the grammar. This prevents operations from breaking silently when a rule is renamed or removed from the grammar: in this case, the call to `addOperation` (i.e., the definition of the operation) will raise an error indicating that the action dictionary must be updated.
2. It checks that the number of formal parameters of each function in the action dictionary is equal to the arity of the corresponding rule in the grammar. This makes the connection between the grammar and the semantic actions more rigid, and prevents operations from breaking silently when the syntax of the language changes.

Note that the second check relies on the fact that branches of an alternation expression are required to have the same arity (as described in Section 2.3). Without this property, a rule like `addExp` would map to a single semantic action that could be called with either one or three arguments, requiring a dynamic check to distinguish between cases. This solution is brittle in the face of changes to the grammar, e.g., if a new case is added or the arity of a case is changed. Ohm's static checks help protect the programmer against a large class of errors that can be caused by such changes.

3. Handling Whitespace in Ohm

If you are familiar with scannerless parsing formalisms, you may have noticed that the calculator we created in the previous section does not support spaces in the input. In most PEG-based parser generators, spaces (and other semantically-irrelevant characters) must be handled explicitly. For example, we could implement space skipping by changing our grammar as follows:

```
exp = addExp space*
addExp = addExp space* "+" mulExp  -- plus
       | addExp space* "-" mulExp  -- minus
       | mulExp
mulExp = ... // similar to addExp
priExp = space* "(" exp space* ")"  -- paren
       | num
num = space* dig+
...
```

Space-skipping expressions make it more difficult for a programmer to "see" the syntax of a language defined by a grammar. Also, the semantic actions of rules that include space-skipping expressions would require additional arguments that would more than likely be ignored. To avoid both of these problems, Ohm handles space skipping implicitly through a simple convention described below.

3.1 Lexical vs. Syntactic Rules

In Ohm, a rule that begins with an upper-case letter is called a *syntactic rule*; all other rules are known as *lexical* rules. In a grammar that contains only lexical rules (like our arithmetic grammar), all whitespace must be explicitly skipped, just like in other PEG-based parser generators.

Inside a syntactic rule, whitespace characters are implicitly skipped before matching an expression. For example, the strings `'ab'`, `' ab'`, and `' a  b'` would all be matched by the following rule:

```
Letters = "a" "b"
```

These same inputs would also be matched by the rule `Letters = letter+`.

```
Arithmetic {
  Exp = AddExp

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

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

  PriExp = "(" Exp ")"  -- paren
         | num


  num = dig+

  dig = "0".."9"
}
```

Figure 3. Modified arithmetic grammar which allows whitespace around operators and expressions.

Figure 3 shows a revised arithmetic grammar that allows whitespace around operators and expressions.

The only difference between this new version of the arithmetic grammar and the original version shown in Figure 1 is that the rules `Exp`, `AddExp`, `MulExp`, and `PriExp` now begin with a capital letter — everything else is unchanged. The names of the semantic actions must also be changed, but they require no other changes to support this new version of the grammar.

With the modified arithmetic grammar, the following inputs will now be accepted:

```
g.match(' 1 +2').succeeded();  // true
g.match('2* 3 ').succeeded();  // true
```

Note that when the start rule (`Exp` in this example) is a syntactic rule, it also consumes trailing whitespace, as shown in the second call to `match()` above.

Ohm's syntactic rules implicitly skip anything that matches the grammar's `space` rule. Every grammar comes with a default implementation of `space`, which matches the traditional whitespace characters (space, tab, line feed, carriage return, etc.). A grammar author can optionally extend or override the built-in `space` rule, e.g., in order to treat comments as whitespace. Ohm's grammar extensibility features are described in the next section.

Sometimes it is necessary to prevent implicit space skipping at a particular point in a syntactic rule. For these cases, Ohm provides the *lexification* operator (`#`). Section 5 describes how we used the lexification operator to implement automatic semicolon insertion in a grammar for the JavaScript language.

4. Extensible Grammars, Modular Semantics

4.1 Grammar Extension

To make it easier to build new languages that are based on existing ones, Ohm supports *grammar extension* using familiar mechanisms from object-oriented programming. An Ohm grammar is analogous to a class, and its rules are analogous to methods. When writing a new grammar, a programmer may *extend* (i.e., inherit from) an existing grammar, and override or extend its rules. If a grammar does not explicitly extend another, then it implicitly inherits from a built-in proto-grammar, which is where `space` and several other generally useful rules are declared. Here is an example of a grammar that inherits from the arithmetic grammar:

```
BetterArithmetic <: Arithmetic { ... }
```

A grammar can declare new rules using the `=` operator, and *override* rules using the `:=` operator. This has the same purpose as the `override` keyword in C#: it enables early error detection by raising a compiler error if the user has made a typo or the original production has been removed in a new version of the parent grammar.

The `+=` operator can be used to add a new case at the beginning of a rule. New cases are inserted at the beginning of a rule because PEGs give preference to earlier branches in an alternation, meaning that earlier expressions in an alternation have more control over the parse. For example, we can use `+=` to add support for some commonly-used constants to the `BetterArithmetic` grammar:

```
BetterArithmetic <: Arithmetic {
  PriExp += const
  const = "pi" -- pi
        | "e"  -- e
}
```

The original definition of `PriExp` was:

```
PriExp = "(" Exp ")"  -- paren
       | num
```

In the `BetterArithmetic` grammar, the line `PriExp += const` is equivalent to having overridden the `PriExp` rule, inserting `const` before the original two cases:

```
PriExp := const
        | "(" Exp ")"  -- paren
        | num
```

However, writing the rule that way would mean that if the definition of `PriExp` in the arithmetic grammar were ever changed, `BetterArithmetic` would have to be kept manually in sync. Using the `+=` operator, the programmer can accurately express an intention to extend (and not replace) the definition in the base grammar.[4]

[4] Note that in PEGs, unlike context free grammars, extending a rule does not imply that the resulting grammar accepts a superset of the inputs accepted by the original. This is due to the semantics of prioritized choice.

If we add this grammar to the same file that contains the original arithmetic grammar ("arithmetic.ohm"), we can use `ohm.grammarsFromFile` to instantiate both grammars:

```
var grammars =
    ohm.grammarsFromFile('arithmetic.ohm');
var g2 = grammars.BetterArithmetic;
```

Now we can use `g2` to match arithmetic expressions that contain the constants `e` and `pi`:

```
g2.match('2 * pi * 8').succeeded();  // true
g.match('2 * pi * 8').succeeded();  // false
```

4.2 Semantic Extension

Recognizing valid expressions is one thing, but we also want to *evaluate* expressions in our `BetterArithmetic` language. If we try to use the `eval` operation we defined previously, Ohm will throw an error: "Cannot use a MatchResult from grammar 'BetterArithmetic' with a semantics for 'Arithmetic' ".

This makes sense: `BetterArithmetic` is a different grammar from `Arithmetic`. Their *syntactic* similarity does not imply that their semantics are the same. In fact, in our original semantics for `Arithmetic`, we could not have specified how to evaluate constants because they weren't even part of the language!

To evaluate expressions in our `BetterArithmetic` grammar, we could create a new semantics (via `g2.createSemantics()`) and define its `eval` operation from scratch, but that would result in lots of duplicated code. Ohm avoids this by providing a mechanism for extending semantics and their operations, in much the same way that you can extend grammars. The only thing required to extend the `eval` operation to our `BetterArithmetic` grammar is to implement actions for the two branches of the `const` rule. Here is how we could do that:

```
var s2 = g2.extendSemantics(s);
s2.extendOperation('eval', {
  const_pi(c) {
    return Math.PI;
  },
  const_e(c) {
    return Math.E;
  }
});
```

The `extendSemantics` method creates a new semantics that inherits operations from another semantics. To extend an operation, we call `extendOperation`, passing an action dictionary. The actions will be combined with the actions from the parent semantics — and if there are duplicates, the new actions override the old ones.

After doing that, `s2`'s `eval` can be used to evaluate expressions containing `pi` and `e`:

```
// Returns 50.26548245743669
s2(g2.match('2 * pi * 8')).eval();
```

In this process, two important checks are made to ensure that the resulting operation is safe:

- **Grammar compatibility.** A grammar's `extendSemantics` method ensures that its argument is a semantics for that grammar or one of its parent grammars.
- **Action arities.** Similarly, a semantics' `extendOperation` method ensures that the resulting action dictionary has the correct arities for all the rules in its associated grammar — whether they were declared in that grammar itself, or inherited, overridden, or extended from the parent grammar. Note that we did not have to implement a new action for `num`, since (a) its arity is unchanged from `Arithmetic`, and (b) its behavior is unchanged (we still want the default pass-through behavior). This check is performed for all of the operations of an extended semantics, including those which were only inherited but not overridden, the first time it is used with a `MatchResult`. (This is necessary because some rules may have been overridden, which could have caused their arities to change.)

5. Case Study: JavaScript

In the past two years, we have found Ohm to be useful in a number of real-world settings. These experiences have shaped the design and helped convince us that the language is sufficiently powerful, while being easy to learn and use. In this section, we describe our experience using Ohm to parse and process JavaScript. This is an interesting case study because:

- JavaScript's *automatic semicolon insertion* is a challenging case for Ohm's implicit whitespace skipping features, and
- the continuing evolution of the language (the 6th version of the specification was released in 2015) offers a real-world example of grammar and semantics extension.

Our JavaScript parser is based on version 5.1 of the ECMAScript Language Specification (known as *ES5*). The language is implemented with a 335-line grammar which lives in the "examples" directory of the Ohm GitHub repository [21].

5.1 Handling Automatic Semicolon Insertion

Like most ALGOL-influenced languages (e.g., C, C++, Java), JavaScript uses the semicolon (`;`) as a statement terminator. However, it allows semicolons to be omitted from the source code in certain situations. The ECMAScript Language Specification [4] details the rules under which semicolons are "automatically inserted" — i.e., the cases in which semicolons are not syntactically required.

In an earlier version of Ohm, this posed a problem for our ES5 grammar. Under certain conditions, ES5 statements

```
ES6 <: ES5 {
  AssignmentExpression<guardIn> += ArrowFunction<guardIn>

  ArrowFunction<guardIn> = ArrowParameters #(spacesNoNL "=>") ConciseBody<guardIn>

  ConciseBody<guardIn> = ~"{" AssignmentExpression<guardIn>  -- noBraces
                       | "{" FunctionBody "}"                -- withBraces

  ArrowParameters = BindingIdentifier
                  | CoverParenthesizedExpressionAndArrowParameterList
  ...
}
```

Figure 4. Fragment of an Ohm grammar implementing support for ES6 arrow function definitions. The full source code is available in the Ohm GitHub repository [21].

may be terminated with a line terminator character rather than a semicolon. However, a line terminator is otherwise considered to be whitespace, meaning it is implicitly skipped over inside syntactic rules, which make up the bulk of our grammar.

The solution to this was to add the *lexification operator* (`#`) to Ohm. The lexification operator can be used to prevent implicit space skipping inside a syntactic rule. For example, consider the `VariableStatement` rule from our ES5 grammar:

```
VariableStatement =
    var VariableDeclarationList<withIn> #sc
```

The expression `#sc` means: apply the `sc` rule, but do not implicitly skip spaces before it. This allows the `sc` rule to explicitly deal with the spaces — making it possible to implement the automatic semicolon insertion rules. If we had just written `sc`, Ohm's implicit space skipping would have taken effect *before* the application of `sc`, potentially consuming a syntactically-significant line terminator.

Without the lexification operator, the alternative would have been to rewrite all of the affected rules as lexical (rather than syntactic) rules. Doing so would have required the insertion of explicit space skipping throughout those rules, causing problems with readability in both the grammar and the semantic actions. The lexification operator, on the other hand, gives us an "escape hatch" that only requires explicit space skipping where it is absolutely necessary.

5.2 Implementing ES6 Arrow Functions

In addition to the ES5 grammar, we have also implemented some features of the ES6 specification. We did this by creating a new `ES6` grammar that extends the `ES5` grammar. Figure 4 shows a fragment of the grammar that implements support for so-called "arrow functions".

Ohm supports *parameterized rules*: rules that take parsing expressions as arguments. The grammar in Figure 4 uses parameterized rules to supply the necessary context for handling ES5 `in`-expressions while avoiding the wholesale duplication of rules seen in the official specification [4].

In addition to grammar extension, we also used Ohm's support for extending semantics in our ES6 example. First, we implemented a `toES5` operation in a semantics for our ES5 grammar. This operation consists of only two semantic actions:

- The `_terminal` action simply returns the original source code for the terminal.
- The `_nonterminal` action invokes `toES5()` on all the child nodes, and concatenates their output.

In other words, for input that matches the ES5 grammar, this operation is simply the identity transformation. However, its purpose is similar to the "template method" design pattern [6]: by invoking `toES5()` on each node in the tree, it is open for extension in a child semantics. For example, we can implement an ES6-to-ES5 compiler by extending the `toES5` operation with semantic actions for the ES6 forms:

```
var s = g.extendSemantics(es5semantics);
s.extendOperation('toES5', {
  ArrowFunction(params, _, arrow, body) {
    var def =
        'function ' + params.toES5() +
        ' ' + body.toES5();
    if (body.mentionsThis()) {
      def += '.bind(this)';
    }
    return def;
  },
  ArrowParameters_unparenthesized(id) {
    return '(' + id.toES5() + ')';
  },
  ConciseBody_noBraces(e) {
    return '{ return ' + e.toES5() + ' }';
  }
});
```

One of the features of ES6 arrow functions is that the `this` keyword is lexically bound (which is not true in regular JavaScript functions). In our implementation, the semantic action for `ArrowFunction` checks if `this` is used anywhere in the function body. This is done using another operation called `mentionsThis`:

```
s.addOperation('mentionsThis()', {
  this(_) { return true; },
  _terminal() { return false; },
  _nonterminal(c) {
    return c.some(n => n.mentionsThis());
  },
  _iter(arr) {
    return arr.some(n => n.mentionsThis());
  }
});
```

Ohm's modular semantics make it easy to define "helper" operations like this inside of a semantics, with no danger of collision with operations defined by another semantics.

5.3 A Note on Performance

Though a detailed analysis of Ohm's performance is outside the scope of this paper, we will briefly discuss the performance characteristics of our modular semantic actions.

We have used our ES5 grammar and the `toES5` operation to parse and re-emit the complete source code of Ohm (approx. 5000 LOC). This represents a kind of worst case for the overhead of our framework, because it visits every single node in the tree (including terminals) and the semantic actions themselves perform very little work.

The results showed that the code generation step — i.e., the execution of the `toES5` operation — accounts for approximately 10% of the total execution time. For comparison, we implemented `toES5` as a recursive function that directly walks the parser's concrete syntax tree, which resulted in a total speedup of 6.8%. Based on these results, and given that we haven't yet put much effort into performance, we believe that the overhead of Ohm's modular semantic actions is unlikely to be a limiting factor in real-world use.

6. Related Work

There are many parser generators, compiler-compilers, and combinator libraries out there, some of which also enforce a separation between grammars and semantic actions. To enable a meaningful comparison with our particular approach, we evaluate each piece or category of related work based on the following scenarios:

- **pure grammar → operation**: A grammar already exists, but it has no semantic actions. The programmer wants to specify what to do with each syntactic construct in the language.
- **the "repetitive semantic actions" problem**: The programmer wants to treat all of the syntactic constructs in the language the same way by default (i.e., with the same semantic action) and provide specialized semantic actions for a few of them.
- **many operations**: The programmer wants to do more than one thing with the grammar, e.g., implement an interpreter and a pretty-printer.
- **modified operation**: An operation (or grammar with semantic actions) already exists; the programmer wants to create a variant of this operation in which certain constructs are handled differently, e.g., make the "+" operator in our calculator multiply rather than add its operands.
- **modified grammar and modified operation**: As in the previous scenario, a grammar with semantic actions already exists. The programmer wants to both create a variant of the original grammar that supports a new syntactic construct (e.g., add a new operator to our arithmetic language) and specify how to handle the new construct.

As we have demonstrated in previous sections, Ohm enables programmers to accomplish all of these tasks without any code duplication.

6.1 Traditional Parser Generators

In this category, we include parser generators in which programmers write semantic actions directly in their grammars. META II [17] is an early example of "a compiler-writing language that consists of syntax equations resembling Backus normal form and into which instructions to output assembly language commands are inserted." A modern, widely-used example of this category of parser generator is Yacc.

In the **pure grammar → operation** scenario, the user of a traditional parser generator may avoid duplicating code by modifying the original grammar in place to add the desired semantic actions. But later, if he needs to do something else with the language — i.e., if he finds himself in the **many operations** scenario — he will have to duplicate the "pure" part of the grammar for each new operation. This creates serious maintenance problems, as future changes to the syntax of the language will require onerous and error-prone modifications to all of the duplicates.

Traditional parser generators do not offer a solution to the **the "repetitive semantic actions" problem**; the programmer is forced to duplicate the same semantic action for most of the rules in the grammar. Even if the programmer places the duplicated semantic action code in a function, he will still have to write identical semantic actions — which are now calls to that function — for most of the rules in the grammar.

To address the **modified operation** scenario, the programmer must duplicate all of the original grammar and most of its semantic actions, and in the **modified grammar and modified operation** scenario, not only the original grammar but *all* of its semantic actions must be duplicated.

6.1.1 Variants with Support for Grammar Inheritance

Some of the parser generators in this category support grammar inheritance, rule overriding, etc. Examples include OMeta, ANTLR, and *Rats!*.

This form of object-oriented extensibility for grammars is not helpful in the **pure grammar → operation** and **many operations** scenarios, since in order to specify a semantic action for a particular syntactic construct, the programmer still has to duplicate the part of the grammar that specifies its syntax. However, it does enable the programmer to avoid some (but not all) code duplication in the **modified operation** scenario: he can write a grammar that extends the original grammar — thus inheriting all of its rules and semantic actions — and override only the rules that correspond to the syntactic constructs that should be handled by semantic actions that are different from those of the parent grammar. In this case, only the parts of the parent grammar that specify the syntax of the overridden rules are duplicated. Object-oriented variants of traditional parser generators also support the **modified grammar and modified operation** scenario with little or no code duplication, but like other traditional parser generators, do not offer a solution for **the "repetitive semantic actions" problem**.

6.1.2 The "AST Solution"

While traditional parser generators are not explicitly designed to keep grammars separate from semantic actions, they do not prevent this separation. It is common for programmers to write a single set of semantic actions that construct an abstract syntax tree (AST). The particular representation of the AST — classes, algebraic data types, etc. — in turn determines the representation of the equivalent of Ohm's operations, as well as the means of extending them. All the extensibility mechanisms of the host language are available to the programmer, but conversely, the programmer must fit their design to the mechanisms provided.

For example, a programmer using an object-oriented language may write AST classes for each syntactic construct in the language. "Operations" may then be implemented as methods in these classes. However, when implementing multiple distinct operations using methods, care must be taken to avoid name clashes or modularity problems. One possible approach is to write support for visitors [6] into the AST classes, and to implement operations as visitors; but no matter the approach chosen, the programmer is left to solve this problem on their own.

Abstract syntax captures the essence of a language, but may elide aspects of the input that are important for some applications. For example, syntax highlighting can depend on fine concrete details of the input, and an AST may not automatically record enough information to highlight effectively. Special care must be taken by the programmer writing supposedly general-purpose parsing code to retain just those not-obviously-significant aspects of the concrete syntax relevant to such applications.

6.2 Parser Generators with Built-In Support for the "AST Solution"

Some parser generators automatically synthesize AST classes from the rules of a pure grammar. As an example, JTB [13] provides this functionality for JavaCC [18] grammars. This design offers the benefits of the "AST solution" (see Section 6.1.2) while saving the programmer the time and effort of building the AST classes manually. Furthermore, the parser generator can anticipate forms of extensibility required in the context of parsing that are not offered directly by the host language itself, helping the programmer express their intent more directly. For example, the parser generator may automatically produce a suitable visitor implementation. However, as with all abstract-syntax-based approaches, important details may be lost in the translation from concrete syntax.

6.3 Parser Combinators

In functional languages, it is common for programmers to use *parser combinator libraries* such as Haskell's Parsec [11] to build recursive-descent parsers. These libraries typically model parsers as functions, and provide higher-order functions (combinators) for composing them.

While a large number parser combinator libraries are available for functional languages, to the best of our knowledge none of them enable the programmer to modularly separate code that recognizes the input (grammars) from code that processes it (semantic actions). There are, however, object-oriented parser combinator libraries that do achieve this form of modularity; we discuss one of them below.

6.3.1 Executable Grammars in Newspeak

Newspeak's Executable Grammars (NSEGs) [1] are a parser combinator library written in a state-of-the-art object-oriented language [3]. Like Ohm, the library separates grammars from semantic actions, and leverages various language features of Newspeak to provide better modularity than parser generators in the previous categories. NSEGs encourage the programmer to write a "pure" grammar as a class, which is customized for each operation via a *mixin* that supplies specific semantic actions.

NSEGs support all of our scenarios without any code duplication, with the exception of the **modified operation** scenario. The problem is that, following transformation by a mixin, a rule's raw CST is no longer available: the semantic value generated by the mixin takes its place. Subsequent mixins that need access to the CST must duplicate the relevant fragment of the grammar to recover it.

Executable Grammars offer extensibility of both grammars and semantic actions by leveraging language features such as mixins and nested classes that are offered by Newspeak. But the fact that these features are not available in most mainstream languages limits NSEG's potential

as a cross-language design. Ohm provides its own mechanisms for supporting modularity and extensibility, which are independent of the host language it happens to be running on. This lets programmers write semantic actions in a language they are already familiar with, and enables access to the libraries available for that language.

While Ohm performs arity checks on semantic actions that help avoid common classes of error, NSEGs offer no equivalent. An NSEG semantic action will fail at parse-time if its arity is incorrect, whereas Ohm will reject an incorrect semantic action dictionary as it is constructed, rather than when it is used. The lack of static checks on semantic actions in NSEGs is especially problematic in cases where the block once had the correct number of arguments, but where the grammar has subsequently evolved. Newspeak's advanced reflection facility [2] could, in a future version of the NSEG library, be used to detect arity errors at grammar construction time, as Ohm does.

6.4 Object Algebras

Ohm's implementation leverages a design pattern that is similar to *object algebras* [12], and as a result our system enjoys the same modularity and extensibility that they provide. In object algebra parlance, an Ohm grammar implicitly specifies (i) an *object algebra interface*, and (ii) a mapping from inputs in concrete syntax to thunks, or *instantiation patterns*, that are parameterized by particular *implementation* of that object algebra interface.

A couple of things are worth noting about the relationship between Ohm and object algebras. First, when the programmer invokes an operation on a match result, Ohm does not eagerly evaluate that operation on its sub-expressions. Which sub-expressions are evaluated, and the order in which they are evaluated, is determined only by the semantic actions that implement the operation. This is a useful form of modularity that the programmer does not get "for free" when using object algebras in an eager language like JavaScript. Second, Ohm's operations can be mutually recursive, i.e., it is possible for an operation `op1` to use another operation `op2` in its implementation, and vice-versa. Support for mutually-recursive operations comes from our *semantics* abstraction, which does not have an analog in object algebras.

7. Conclusions and Future Work

In this paper, we have described the design of the Ohm parser generator and demonstrated the benefits of its strict separation of syntax and semantics. As discussed in Section 5, Ohm has already proven useful in real-world scenarios.

While we have found Ohm's expressive power to be sufficient for most tasks, several users have requested better support for parsing context-sensitive languages like HTML and Python. We plan to investigate whether a solution like *parsing contexts* [10] (as implemented in PetitParser [16]) could be adapted to Ohm.

We also plan to implement support for incremental parsing [7], so that small changes to a previously-parsed input do not require the entire input to be reparsed. This would make Ohm more suitable for interactive use cases with potentially large inputs — such as syntax highlighting in an IDE.

Finally, as the language stabilizes, we would like to create and curate a publicly-available collection of Ohm grammars that anyone can contribute to and use. The complete separation of grammars and semantic actions in Ohm makes its grammars truly "cross-platform," which means that programmers can use the same grammar with different Ohm implementations.

Acknowledgments

The authors would like to thank Gilad Bracha, Jonathan Edwards, Saketh Kasibatla, Meixian Li, Todd Millstein, Yoshiki Ohshima, Marko Röder, Tjis van der Storm, Laurie Tratt, and the anonymous reviewers for feedback on this work and earlier drafts of this paper.

References

[1] G. Bracha. Executable grammars in Newspeak. *Electronic Notes in Theoretical Computer Science*, 193: 3–18, 2007.

[2] G. Bracha and D. Ungar. Mirrors: design principles for meta-level facilities of object-oriented programming languages. In *Proc. OOPSLA*, 2004.

[3] G. Bracha, P. von der Ahé, V. Bykov, Y. Kashai, W. Maddox, and E. Miranda. Modules as objects in Newspeak. In *Proc. ECOOP*, pages 405–428. Springer, 2010.

[4] Ecma International. *ECMAScript 2015 Language Specification*. Geneva, 6th edition, June 2015.

[5] B. Ford. Parsing expression grammars: A recognition-based syntactic foundation. In *ACM SIGPLAN Notices*, volume 39, pages 111–122. ACM, 2004.

[6] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley, 1995.

[7] C. Ghezzi and D. Mandrioli. Augmenting parsers to support incrementality. *J. ACM*, 27(3):564–579, July 1980.

[8] R. Grimm. Better extensibility through modular syntax. *ACM SIGPLAN Notices*, 41(6):38–51, 2006.

[9] S. C. Johnson. *Yacc: Yet another compiler-compiler*, volume 32. Bell Laboratories Murray Hill, NJ, 1975.

[10] J. Kurš, M. Lungu, and O. Nierstrasz. Top-down parsing with parsing contexts. In *Proc. IWST*, 2014.

[11] D. Leijen and E. Meijer. Parsec: Direct style monadic parser combinators for the real world. Technical report, 2001.

[12] B. C. d. S. Oliveira and W. R. Cook. Extensibility for the masses: Practical extensibility with object algebras. In *Proc. ECOOP*, pages 2–27. Springer, 2012.

[13] J. Palsberg, K. Tao, and W. Wang. Java Tree Builder website, 1997. URL `http://compilers.cs.ucla.edu/jtb/`.

[14] T. Parr and R. Quong. ANTLR: A predicated LL(k) parser generator. *Software—Practice and Experience*, 25(7):789–810, 1995.

[15] P. Rein, R. Hirschfeld, and M. Taeumel. Gramada: Immediacy in programming language development. In *Proc. Onward!* ACM, 2016.

[16] L. Renggli, S. Ducasse, T. Gîrba, and O. Nierstrasz. Practical dynamic grammars for dynamic languages. In *Proc. DYLA*, 2010.

[17] D. V. Schorre. META II: a syntax-oriented compiler writing language. In *Proc. ACM National Conference*. ACM, 1964.

[18] S. Viswanadha and S. Sankar. JavaCC website. URL `https://javacc.java.net/`.

[19] A. Warth. *Experimenting with Programming Languages*. ProQuest, 2009.

[20] A. Warth and I. Piumarta. OMeta: an object-oriented language for pattern matching. In *Proc. DLS*, pages 11–19. ACM, 2007.

[21] A. Warth, P. Dubroy, and T. Garnock-Jones. Ohm/JS repository, 2014. URL `https://github.com/cdglabs/ohm`.