2008

Monadic parsing using JavaScript

Axel-Tobias Schreiner

Monadic Parsing using JavaScript

Axel T. Schreiner, Department of Computer Science, Rochester Institute of Technology

Abstract

Monad is a class in Haskell which is fundamental to encapsulating side effects. A monadic type can be used, e.g., to maintain and manipulate state alongside another computation or to bypass sequential execution and recover from failure. A significant problem domain is parsing: support for monadic parsers exists for Haskell, Python, and other languages.

This web page describes monadic LL(n) parsing with JavaScript, complete with a base class for monadic classes which wrap state functions, a notation to embed monadic computations in JavaScript (i.e., an equivalent to the `do` notation in Haskell), a preprocessor to translate the notation into JavaScript, a scanner generator based on regular expressions, a factory for classes to represent parse trees, and the implementation of a little language with exception handling as another example of a monadic computation. The preprocessor is implemented using the monadic parser which it supports.

Technical Information

All source code in this web page is live and can be edited and executed interactively. A number of examples have been set up for execution in this web page. They are usually followed by suggestions for further investigations which involve editing and re-executing the examples. The page has been tested with numerous browsers in Mac OS X and Windows XP -- only Internet Explorer had issues with preprocessing the examples. (Details are recorded in the comments in this page.)

The source code is organized in namespaces, i.e., global collections; each editable area is labelled with the namespace it belongs to and has a reset button to restore the original content. Two more pages are opened behind this page which contain the remaining code of two namespaces (`JSM`, the preprocessor for the monadic notation, and `Mini`, the implementation of a small programming language) of which only excerpts are presented in this web page. Using the links in below the original content of each namespace can be displayed on a separate web page.

Monadic notation needs to be translated to Javascript before it can be executed; this is accomplished using the preprocess button following each editable area with monadic notation. Any preprocess button can be (ab-)used to translate arbitrary code inserted into the preceding editable area; the reset button will restore the original content.

A read-only (preprocess or output) area can be expanded or contracted by clicking into the area. The following buttons can be used preprocess all code (except where the preprocessed source is part of the web page to begin with), execute all JavaScript examples, interpret all little language examples, and expand the size of all read-only areas.

preprocess all execute all interpret all

Introduction

In Haskell a data type is a set of values. A data type can be defined as an instance of one or more classes; a class declares polymorphic operators and methods which a type instance must implement. *Monad* is a class with operations which control sequential execution; the *do* notation is a very convenient way to express computations in a seemingly imperative style as long as they involve values from types which are instances of *Monad*. Implementing the *Monad* operations makes it possible to abandon sequential execution, e.g., there can be a reaction to failure, and state can be maintained separately from sequential execution, e.g., to trace operations, support a form of assignment, or perform input and output. Specifically, Haskell uses the *IO* monad to separate stateful input/output from stateless, "pure" functional programming.

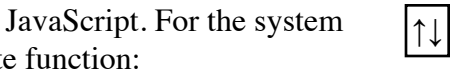
Many imperative languages such as C# , Groovy , Java , JavaScript , Python , Ruby , and of course all variants of Lisp , support functions as first-order values. Some of these languages, e.g., JavaScript and Lisp, are dynamically typed, others, like Haskell or C#, support some form of type inference; either variant of typing greatly simplifies the use of functions. However, but for Haskell, most languages do not integrate monadic types with special language constructs.

If a programming language supports assignment, input/output, and some kind of global storage, there is no pressing need for monads because state can be maintained rather explicitly. However, this web page shows that monads can be created given only functions as first-order values. The web page illustrates that -- at least in the area of parsing -- monads are a very useful way of structuring computation. Specifically, this web page shows how to program recursive descent parsers directly from LL(n) grammars based on an infrastructure which requires only regular expressions and functions as first-order values. It also extends JavaScript with a notation similar to Haskell's *do* notation which can be viewed as the input language for a parser generator, but which can be used for other monadic computations as well. There is a short discussion how the notation is used to convert itself into JavaScript, i.e., how the parser generator is implemented. Finally, there is an implementation of an interpreter for a very small, conventional programming language which serves as an example for other monadic computations.

The web page provides tools and examples for those who need to implement small languages using JavaScript, and it suggests by example how the tools can be quickly implemented in other languages which support regular expressions and functions as first-order values.

What's in a Monad?

This section describes `Monad`, a base class for monadic classes in JavaScript. For the system described here a monadic value is an object which contains a state function:



Monad

```
function Monad (stateFunction) { this.stateFunction = stateFunction; }
```

reset

Given a monadic value, its state function can be applied to a state value:

Monad

```
Monad.prototype.apply = function (state) {
  return this.stateFunction(state);
};
```

reset

Unlike Haskell, JavaScript is a dynamically typed language, i.e., neither the type of `state` nor the return type of the state function have to be specified; therefore, `Monad` can provide the functionality of all state-function-based monadic classes. By convention in this system, on success a state function will return a collection with two properties, the new `state` and the `value` encapsulated by the monadic value which contains the state function; on failure a state function will return a collection with a `fail` property, e.g., with an error message. With this convention `Monad` combines the capabilities of Haskell's *Either* and state-function-based monads.

Given two monadic values, `orElse` creates a new monadic value in the same class as the receiver. The new value contains a state function which will apply the receiver's state function or -- only in case of failure -- the state function contained in the argument value for `orElse`:

`Monad`

```
Monad.prototype.orElse = function (b) {
  var a = this;                          // for closure
  return new this.constructor(           // receiver's class
    function (state) {
      var result = a.apply(state);
      return 'fail' in result ? b.apply(state) : result;
    }
  );
};
```

reset

Loosely speaking, monadic values "are" state functions and `orElse` combines them for alternative execution. It should be noted that either state function is applied to the same incoming state. In the terminology of Haskell, `orElse` is the *mplus* operation of the *MonadPlus* class.

Intuitively, the *bind* operation, denoted as `>>=` in Haskell's *Monad* class, combines two monadic values, i.e., state functions, for sequential execution (and creates a scope for a result value). However, sequential execution has to be controllable: it must be guaranteed that the first state function is executed first, and it should be possible to suppress executing the second state function if the first one fails. Additionally, and unlike `orElse`, if both state functions succeed the final state and value should depend on both functions.

Therefore, the method `andThen` does not combine two monadic values directly. Instead, it accepts as an argument a function which is expected to accept a `value` property produced by the receiver's state function and return the monadic value which `andThen` is to combine with the receiver:

`Monad`

```
Monad.prototype.andThen = function (b) {
  var a = this;                           // for closure
  return new this.constructor(            // receiver's class
    function (state) {
      var result = a.apply(state);
      return 'fail' in result ? result
        : b(result.value).apply(result.state);
    }
  );
};
```

reset

The method `andThen` returns a monadic value in the same class as the receiver containing a state function which first applies the receiver's state function. If successful, the `result.value` is used to produce the second monadic value and its state function is applied to the `result.state`. Loosely speaking the final value results from composing both state functions, the incoming state is sent only to the first state function, and its result state is sent to the second state function.

`Monad.subclass` is a convenience function to create constructors for new monadic classes which inherit the methods described thus far. (Strictly speaking, there is no inheritance among classes in JavaScript; however, this web page uses the terminology of Java. In this parlance `Monad.subclass` is a class method -- which is not inherited by the new monadic classes.)

Monad

```
Monad.subclass = function () {
  var result = function () {          // the new constructor
    Monad.call(this, arguments[0]);  //   chained to Monad's constructor
  };

  result.prototype = new Monad();
  delete result.prototype.stateFunction;
  result.prototype.constructor = result;
```

reset

Monad

```
  // ... method definitions ...
  return result;
};
```

reset

`Monad` is the superclass of all monadic classes. Therefore, the new constructor `result` is chained to the `Monad` constructor and a `Monad` object is set up as the `prototype` of the new class. The purpose of the `prototype` is to inherit `andThen`, `apply` and `orElse`; therefore, the `stateFunction` is deleted from the `prototype`. Finally `result` is added as `constructor` to the `prototype` and returned.

A monadic class should contain some monadic values. Therefore, `subclass` creates a few class methods for the new class which create monadic values. `succeed` creates an instance of the new class for its argument value, i.e., it returns a monadic value containing a state function which will return a collection with the argument value of `succeed` and the incoming state:

Monad

```
result.succeed = function (value) {
  return new result(
    function (state) {
      return {value: value, state: state, toString: result.dump };
    }
  );
};
```

reset

`dump` serializes a collection and is connected as `toString` method for the result of the state function defined for `succeed`, but it can also be used as a class method to display its argument:

Monad

```
result.dump = function dump () {
  var s = '',
    arg = arguments && arguments.length ? arguments[0] : this;
  if (arg == null) return 'null';
  if (typeof arg == 'undefined') return 'undefined';
  if (typeof arg != 'object') return arg.toString();
  for (var key in arg)
    if (arg.hasOwnProperty(key) && key != 'toString')
      s += ', '+key+': '+dump(arg[key]);
  return s ? '{'+s.substring(2)+'}' : '';
};
```

reset

The class method `fail` creates a monadic value with a state function which will return a collection with the argument of `fail`:

Monad

```
result.fail = function (message) {
  return new result(
    function (state) { return { fail: message, toString: result.dump }; }
  );
};
```

reset

`succeed` and `fail` are the *return* and *fail* operations required by Haskell's *Monad* class. `fail` is also the operation *mzero* of *MonadPlus*.

Finally, two more kinds of monadic values will turn out to be useful and are defined as value of a class variable and results of a class method, respectively. `get` is a monadic value containing a state function which returns the incoming state as both, `value` and `state`; it is used to expose the current state within a monadic computation:

Monad

```
      result.get =
        new result(
          function (state) {
            return { value: state, state: state, toString: result.dump };
          }
        );
```

reset

`put` returns a monadic value containing a state function which ignores the incoming state and returns the arguments of `put` as the new `value` and `state`; it is used to set the current state from within a monadic computation:

`Monad`

```
      result.put = function (value, state) {
        return new result(
          function () {
            return { value: value, state: state, toString: result.dump };
          }
        );
      };
```

reset

To simplify debugging, all result values are connected to `dump`.

Monad axioms

Operations of *Monad* and *MonadPlus* should satisfy certain axioms which can now be tested using `Monad`. The tests are cumulative and can be edited and executed below. With a stand-alone implementation of JavaScript such as *Rhino* or *SpiderMonkey* the examples can be executed interactively once the `Monad` definition from the previous section is loaded.

`Axioms` is a monadic class and `m` is a monadic value for which the axioms will be tested; `Axioms` is also used as a namespace to avoid global clutter:

`Axioms`

```
    var Axioms = Monad.subclass();
    with (Axioms) {
      Axioms.m = succeed('hello');
      dump(m);
    }
```

execute reset

Used as a class method above, `dump` shows that `m` contains a state function which will return the argument `'hello'` used to create `m` and the incoming state. JavaScript cannot show that the free variable `value` of the state function has in fact closed over the argument given to `succeed` when `m` was created.

Implicitly used to convert to a string below, `dump` shows that the result of applying the state

function contained in `m` combines `'hello'` and the incoming state `'s'`.

Axioms

```
with (Axioms) {
  m.apply('s');
}
```

execute reset

Try this: `fail('message')`, `get`, and `put('value', 'state')` can be used to obtain other monadic values. Change `m` above, or introduce additional variables, and examine the results above and when testing the axioms below. □

Loosely speaking, if one considers monadic values as a set with an operation `andThen` the function `succeed` has to act as a right unit:

Axioms

```
with (Axioms) {
  m.andThen(succeed).apply('s');
}
```

execute reset

This shows that the two monadic values `m` and `m.andThen(succeed)` exhibit the same behavior. However, they contain very different state functions:

Axioms

```
with (Axioms) {
  dump(m.andThen(succeed));
}
```

execute reset

Try this: Note that the results of the two invocations of `apply` equal each other, independent of which monadic value you bind to `m` because `succeed` acts as a right unit for `andThen`. Also note that the state function immediately above does not change because it is the result of `andThen` itself. □

Similarly, a monadic value constructed with `succeed` acts as a left unit. Given some function `f` which returns a monadic value and some argument `'hello'`:

Axioms

```
with (Axioms) {
  Axioms.f = function (v) { return get; }
  f('hello').apply('world!');
}
```

execute reset

The monadic value `f('hello')` exhibits the same behavior above as the monadic value resulting from combining `succeed('hello').andThen(f)` below:

Axioms

```
with (Axioms) {
  succeed('hello').andThen(f).apply('world!');
}
```

execute reset

Try this: Again, the monadic value in the definition of `f` above can be changed to check the behavior for another function. Define `f` so that the result involves `hello world!`. □

The third axiom requires `andThen` to be an associative operation. Continuing the example here is an illustration which involves a monadic value constructed with `succeed`:

Axioms

```
with (Axioms) {
  Axioms.g = function (v) { return succeed(v+' world!'); }
  m.andThen(f).andThen(g).apply('hello');
}
```

execute reset

The original definition of `f` uses `get` which copies the incoming state `'hello'` as value and `g` uses `succeed` to append `' world!'` to it. The chain above is executed from left to right, but the functions can be combined to change this:

Axioms

```
with (Axioms) {
  m.andThen(function (x) { return f(x).andThen(g); }).apply('hello');
}
```

execute reset

The example shows that (at least for these specific monadic values and functions) the behavior does not depend on whether `m` is first combined with `g` and the result is combined with `f` (association from the left), or `m` is combined with the result of combining `g` with `f` (association from the right).

Try this: Change the definitions of `m`, `f`, and `g` to investigate other cases for the third axiom. □

MonadPlus operations are also expected to satisfy certain axioms: *mzero* should act as a right and

left zero in combinations with >>= and *mplus*, i.e., `fail`, `andThen` and `orElse` should act just like *false* combined with preemptive *and* and *or* operations in a programming language.

x and *false* is *false*:

Axioms

```
with (Axioms) {
  get.andThen(function (v) { return fail('fail'); }).apply('hello');
}
```

execute reset

x or *false* is *x*:

Axioms

```
with (Axioms) {
  get.orElse(fail('fail')).apply('hello');
}
```

execute reset

false and *x* is *false*:

Axioms

```
with (Axioms) {
  fail('fail').andThen(g).apply('hello');
}
```

execute reset

false or *x* is *x*:

Axioms

```
with (Axioms) {
  fail('fail').orElse(get).apply('hello');
}
```

execute reset

Try this: Introduce a variable `x` in each axiom and bind different monadic values to `x` to confirm the axioms for other values. Change the function `g` and check again. □

It should be stressed that the examples do not *prove* that the methods and values inherited from `Monad` satisfy the axioms. The examples just illustrate that the axioms are observed for a specific

monadic value and for the function `g` defined earlier; many values and functions can be inserted to check again.

What's in a Parser?

Ideally, a parser examines an input string and produces a value, e.g., the input string might contain an arithmetic expression and the value is a tree representing the expression, or even the value of the expression. A parser is often unable to complete the task -- it might understand just an initial part of the input or even fail completely. Therefore, it makes sense to require that a parser function accept input and return a value and the remaining input if successful, and a string with an error message if not. In other words a parser function is a state function and the state is the input to be examined.

This section describes a class `Parser` with monadic values containing parser functions:

Parser

```
var Parser = Monad.subclass();
```

reset

This section owes much to Hutton's excellent book . He starts with parsers which accept a single character or even nothing at all and builds up to parsers which accept numbers or identifiers optionally surrounded by white space. However, JavaScript supports regular expressions which simplify the description of low-level building blocks for complex parsers. For example, the following collection suffices to describe the pieces which can be combined to form arithmetic expressions:

arithmetic

```
var arithmetic = {
  skip:    /\s+/,     // white space
  number:  /[0-9]+/,  // digit sequence
  symbol:  /./,       // operators, parentheses, etc.
  eof:     /$/        // end of file
};
```

reset

A collection with a few more properties can already describe a small programming language:

language

```
var language = {
      skip:    /^(\s|\/\/.*)+/,            // space and comments
      number:  /^[0-9]+(\.[0-9]*)?/,       // decimal value
      symbol:  /^(<=|>=|<>|.)/,            // operators and comparison
      word:    /^[a-zA-Z_][a-zA-Z_0-9]*/,  // identifier et al.
      quoted:  /^"([^"\\\n]|\\.|\\\n)*"/,  // Java string
      eof:     /^$/                        // end of file
    };
language.word.reserved = ['if', 'else', 'while'];  // reserved words
```

reset

The property `skip` has special significance as discussed below; all other property names can be chosen at will. Every property has a regular expression as a value and all but `skip` can themselves have a property `reserved` with a list of exceptions to indicate matches with a special meaning. A parser always examines the initial part of the input; therefore, the regular expressions are more efficient if they are anchored with `^`.

`reserved` allows to create parsers for specific symbols and for all other symbols matching a pattern. For example, many programming languages use the same conventions for user-defined identifiers and language-specific reserved words. Therefore, the regular expression specified for a property like `word` is assumed to match both kinds of input. If a match of `word` is contained in `word.reserved` it is accepted only by a parser generated for a reserved word, not by the parser which accepts anything else matched by the regular expression specified for `word`.

`Parser.Factory` is a class of objects which can produce monadic `Parser` values. A factory object is constructed with a collection of properties as described above. Optionally, a trace flag can be specified:

Parser

```
    Parser.Factory = function (table, trace) {
      var self = this;                          // for closure
      if (table.skip)                           // create skip parser
        self.skip = new Parser(
          function (input) {
            return self.scan(input, table.skip, 'skip');
          });
```

reset

Parser

```
      // create other parsers if any...
    };
```

reset

The actual analysis of input, i.e., the body of the parser function, can be delegated to a common method `scan` which accepts input and a regular expression and returns a collection with the matched text as `value` and the remaining input as `state`:

Parser

```
    Parser.Factory.prototype.scan = function (input, re, e) {
      if (typeof input == 'string')                         // initialize
line count
        input = {lno: 1, text: input};
      var m = re.exec(input.text);
      if (!m || m.index)
        return this.error(input, 'expecting "'+e+'"');
      var nls = m[0].match(/\n/g);                          // # newlines in
match
      return {value: m[0],
              state: {lno: input.lno + (nls ? nls.length : 0),
                      text: input.text.substring(m[0].length)},
```

reset

The original input can be a string, but the remaining input will be a collection with the remaining string as `text` and the line number where this string begins as `lno`. If there is no match, or if the match is not at the beginning of the input, the result is an error message which includes the current line number in the input:

Parser

```
Parser.Factory.prototype.error = function (input, message) {
  return { fail: '('+input.lno+') '+message, toString: Parser.dump };
};
```

reset

Either result is connected to `dump` to simplify debugging, for example:

Scanners

```
var Scanners = {};  // namespace
with (Scanners) {
  Scanners.af = new Parser.Factory(arithmetic);
  af.skip.apply(' \n foo bar');
}
```

execute reset

Try this: The pattern `arithmetic.skip` matches one or more white space characters. Remove the leading white space from the input to see an error message when `skip` fails. Note the different line numbers.

Try this: Change `arithmetic` so that a newline character is recognized as a `symbol` rather than white space, i.e., the example above should fail as soon as the single leading space is removed from the input. Apply `symbol()` to recognize the leading newline. □

The factory instance variable `skip` references a parser. For every other property in the collection a method is added to the factory which will create a parser. If there is a `skip` property, these parsers will discard a match before they consider the rest of their input:

Parser

```
      for (var p in table)                // all other properties
        if (p != 'skip') {
          self[p] = (function (p) {       // property name for closure
            return function (expected) {  // generator method
              if (!expected || !self[p].reserved
                  || expected in self[p].reserved)
                return new Parser(        // created by generator method
                  function (input) {      // parser function
                    if (self.skip) {      // discard match of skip if any
                      var skipped = self.skip.apply(input);
                      if (!('fail' in skipped))
                        input = skipped.state;
                    }                     // match property's expression
                    var result = self.scan(input, table[p], p);
                    if (!('fail' in result))
                      if (expected) {     // must equal argument if any
                        if (result.value != expected)
                          result = self.error(input, 'expecting
"'+expected+'"');
                      } else              // or must not be in reserved list
if any
                        if (self[p].reserved && result.value in
self[p].reserved)
                          result = self.error(input, '"'+result.value+'" is
reserved');
                    if (trace && !('fail' in result))
                      print('('+result.state.lno+') '+p+': '+result.value);
                    return result;
                  });
              else                        // argument must be in reserved list
if any
                throw new Error('"'+expected+'" is not reserved');
            };
          })(p);
          if (table[p].reserved) {        // create reserved list
```

reset

For example, the parser factory created from the `arithmetic` collection is sufficient to deal with arithmetic expressions:

Scanners

```
    with (Scanners) {
      Scanners.a = af.number().apply(' 12 + 34 ');
      Scanners.b = af.symbol('+').apply(a.state);
      af.number(34).apply(b.state);
    }
```

execute reset

Try this: Each parser picks one item from the input; invoke `print` to display the successive values above. ☐

If the `scan` method finds a match the result value can be filtered with an argument specified to the

generator method.

Try this: Change the argument of `number` to see the third parser fail. Then change the original input so that the parser succeeds again. ☐

If there is a `reserved` list for a property the corresponding method creates a parser only if the argument, if any, is in the list. The `language` collection includes reserved words:

Scanners

```
with (Scanners) {
  Scanners.lf = new Parser.Factory(language);
  Scanners.lf.identifier = lf.word();
  Scanners.lf.number = lf.number();
  Scanners.lf.ifToken = lf.word('if');
  Scanners.lf.emptyString = lf.quoted('""');

  Scanners.a = lf.ifToken.apply(' if a // comment \n "" + 10 ');
  Scanners.b = lf.identifier.apply(a.state);
  Scanners.c = lf.emptyString.apply(b.state);
  Scanners.d = lf.symbol('+').apply(c.state);
  Scanners.e = lf.number.apply(d.state);
  lf.eof().apply(e.state);
}
```

execute reset

Try this: Again, invoke `print` to display how the successive items in the original input are accepted; i.e., print `a`, `b`, `c`, `d`, and `e`. ☐

Parsers are objects which can be applied to accept input. Parsers can be created by generator methods of a parser factory. The example shows that frequently used parsers can be saved, e.g., as instance variables of the parser factory; the instance variable can even replace the generator method.

If there is no `reserved` list for a property, a parser can be generated without an argument to accept any input matching the corresponding regular expression, or it can be generated with an argument to accept only input which must match the regular expression and must equal the argument. If there is a `reserved` list, a parser can be generated with an argument from the list to accept input equal to the argument; if it is generated without an argument it accepts any match of the regular expression which is not in the `reserved` list. In any case, if there is a `skip` expression, input matching `skip` is first discarded.

In summary, regular expressions are used to partition input, `skip` indicates input to be discarded, generator arguments restrict what partitioned input is acceptable, and each `reserved` list describes special cases to be considered once input has been partitioned.

Combining parsers

This section discusses how simple parsers can be combined to create parsers which accept input with a more complicated structure. Consider a grammar for arithmetic expressions:

```
term:    number | '(' sum ')';
product: term '*' product | term '/' product | term;
sum:     product '+' sum | product '-' sum | product;
```

A `Parser.Factory` based on the `arithmetic` collection introduced in the preceding section produces parsers for *`number`* and the literal operator symbols used in this grammar. Parsers are monadic values and can be combined with `andThen` for sequential execution and `orElse` for alternatives, i.e., those two operations are sufficient to construct parsers for the nonterminals `term`, `product`, and `sum` in this grammar, simply by translating the grammar:

Expr

```
var Expr = {};  // namespace
with (Expr) {
  Expr.af = new Parser.Factory(arithmetic);

  Expr.term =            af.number()             .orElse(
                         af.symbol('(')       .andThen(
    function () { return sum              .andThen(
    function () { return af.symbol(')'); }       ); } ) );

  term.apply(' 10 ;');
}
```

execute reset

Try this: Use a different *`number`* as input to `term`. Then try to input `(10)`. ☐

Each terminal symbol of the grammar such as *`number`* or `'('` is translated into a parser generated from a pattern collection by `Parser.Factory`.

Each nonterminal symbol such as `term` is translated into a parser combined from other parsers: a sequence of items in the grammar is combined using `andThen`, alternatives are combined using `orElse`. The example above shows that the parsers can often be applied before all other parsers have been defined.

Expr

```
with (Expr) {
  Expr.product =          term                .andThen(
    function () { return af.symbol('*') .andThen(
    function () { return product; }             ); } )   .orElse(
                          term                .andThen(
    function () { return af.symbol('/') .andThen(
    function () { return product; }             ); } ) .orElse(
                          term                           ) );

  Expr.sum =              product             .andThen(
    function () { return af.symbol('+') .andThen(
    function () { return sum; }                 ); } )   .orElse(
                          product             .andThen(
    function () { return af.symbol('-') .andThen(
    function () { return sum; }                 ); } ) .orElse(
                          product                        ) );

  term.apply(' (1 + 2*3) ; ');
}
```

execute reset

Try this: Input some other arithmetic expressions. Also, specify an invalid expression such as `-23`. Change the definition of `term` to permit a single (or even more than one) minus sign. ☐

With a preprocessor-supported notation as described below it will be much easier to specify this nest of function calls. Nevertheless, the result is a functioning parser for arithmetic expressions -- it produces a `value` and the remaining input for a correct phrase as shown above, and an error message in case of failure:

Expr

```
with (Expr) {
  Expr.expr =             sum       .andThen(
    function () { return af.eof(); }      );

  expr.apply(' (1 + 2*3) \n ; ');
}
```

execute reset

`expr` expects that the input string contains only numbers, operators, and parentheses in the proper order, and optionally white space, nothing else. In the example above the expression is followed by a semicolon on a second line to demonstrate that an error message contains the current line number.

Try this: Remove the semicolon from the input to see `expr` succeed. Input other expressions. ☐

It should be noted that the order of the alternatives is important. `orElse` does not implement backtracking, it only activates the second parser if the first one fails. For example, the alternatives of `sum` could be reordered:

```
sum:     product | product '+' sum | product '-' sum;
```

which translates into

Expr_badSum

```
with (Expr) {
  Expr.badSum =          product                          .orElse(
                         product              .andThen(
    function () { return af.symbol('+') .andThen(
    function () { return sum; }                  ); } ) .orElse(
                         product              .andThen(
    function () { return af.symbol('-') .andThen(
    function () { return sum; }                  ); } )        ) );

  af.error = function (input, message) {
    var result = Parser.Factory.prototype.error(input, message);
    print(result); return result;
  };

  badSum.apply(' 1 + 2 ');
  // delete af['error']
}
```

execute reset

Try this: How much of the input above does `badSum` accept? What happens if the input is enclosed in parentheses? ☐

The `error` method is (temporarily) overridden above so that it produces output as soon as there is any failure. Execution with the original input shows that only the operators of `product` are searched. The two failures happen when `product` tries to find a multiplication operator before it settles for a simple `term`. While `badSum` succeeds, it does not accept the entire expression because the first alternative succeeds with a simple `product`. Therefore, when a grammar is translated into parsers, the longer alternatives have to be specified first.

The example suggests that it is inefficient when a failure happens in the middle of an alternative: `orElse` starts over with the original input, i.e., in this case `1` is parsed by `number` and `term` a total of three times before `product` finally succeeds. The grammar can be changed to avoid this:

```
betterProduct: term mulDivs;
mulDivs:       '*' betterProduct | '/' betterProduct | /* empty */;
```

An empty alternative succeeds without accepting input, i.e., without changing state:

Expr

```
with (Parser) with (Expr) {
  Expr.betterProduct =   term    .andThen(
    function () { return mulDivs; }      );

  Expr.mulDivs =         af.symbol('*') .andThen(
    function () { return betterProduct; }       )   .orElse(
                         af.symbol('/') .andThen(
    function () { return betterProduct; }       ) .orElse(
                         succeed('')                    ) );

  betterProduct.apply(' 2 * 3 ');
}
```

execute reset

`with (Parser)` is needed in this code because `Parser` provides the method `succeed`.

Try this: Insert `badSum` above and combine it with `betterProduct`. Does this improve performance? Does it reduce the number of failures if the input is enclosed in parentheses? ☐

Extended BNF, pioneered by Nicklaus Wirth for Pascal , introduces constructs to describe a grammar and avoid recursion. A popular style for the constructs is usually used in Internet RFCs : items can be grouped with parentheses, optional items are marked with a suffix `?`, and the suffixes `*` and `+` indicate that an item can be repeated zero and one or more times, respectively. The grammar for arithmetic expressions and the translation to parsers could be changed as follows:

```
betterSum: product summands?;
summands:  ('+' product | '-' product)+;
```

This translates into:

Expr

```
with (Expr) {
  Expr.betterSum =       product        .andThen(
    function () { return summands.optional(); } );

  Expr.summands =        af.symbol('+') .andThen(
    function () { return product; }             ) .orElse(
                         af.symbol('-') .andThen(
    function () { return product; }             )        ).some();

  betterSum.apply(' 2 + 3 - 4 ');
}
```

execute reset

Try this: Add more summands to the input and observe the resulting value. ☐

Often, the parentheses need not be translated explicitly; here they are implied by the fact that method application is left-associative. The implementation uses two of the `Parser` methods

`optional`, `some`, and `many` corresponding to the suffixes `?`, `+`, and `*`, respectively, which can be applied to any `Parser` value:

Parser

```
      Parser.prototype.optional = function (value) {
        with (Parser)
          return this.orElse(succeed(arguments && arguments.length > 0 ? value :
'' ));
      };

      Parser.prototype.some = function () {
        var self = this; // for closure
        with (Parser)
                                  return self
.andThen(
          function (fromSelf) { return self.many()                        .andThen(
          function (fromMany) { return succeed([fromSelf].concat(fromMany)); } )
} );
      };

      Parser.prototype.many = function () {
```

reset

Given a parser, `optional` creates a new parser with a parser function which will execute the receiver's parser function and return the result if successful; otherwise it will return the argument `value` (or an empty string) and the original input as `state`.

Given a parser, `some` creates a new parser with a parser function which will execute the receiver's parser function one or more times and return a list with the results as `value` and the remaining input as `state`; the parser function fails if the receiver function does not succeed at least once.

Given a parser, `many` creates a new parser with a parser function which tries to execute the receiver's parser function one or more times and return a list with the results as `value` and the remaining input as `state`; the parser function will always succeed but the resulting `value` list is empty and the input `state` is unchanged if the receiver's parser function does not succeed at least once.

The implementation of `some` uses the fact that `andThen` combines a parser and a function and passes the `value` resulting from successful execution of the receiver's parser function as argument to the function. In the chain of `andThen` above the function definitions are nested so that all parameter scopes extend to the end of the innermost function, i.e., any parameter can be used from the point where it is introduced and in all the nested functions.

The implementations of the methods above suggest that a parser function need not always return a string as `value`. In fact, `some` and `many` arrange to return lists and `optional` can arrange to return an arbitrary value. This can be exploited, e.g., to interpret an arithmetic expression as it is parsed. Here is a grammar expressed with extended BNF:

```
term:    number | '(' sum ')';
product: term ('*' term | '/' term)*;
sum:     product ('+' product | '-' product)*;
expr:    sum eof;
```

This translates into the following interpreter:

Eval

```
      var Eval = {};  // namespace
      with (Parser) with (Eval) {
        Eval.af = new Parser.Factory(arithmetic);

        Eval.term =             af.number()                 .orElse(
                                af.symbol('(')           .andThen(
          function ()  { return sum                   .andThen(
          function (s) { return af.symbol(')') .andThen(
          function ()  { return succeed(s); }          ); } ); } ) );

        Eval.product =           term
.andThen(
          function (l)  { return af.symbol('*')  .andThen(
          function ()   { return term       .andThen(
          function (r)  { return succeed(
             function (x) { return Number(x) * Number(r); }
                                    ); }        ); } ) .orElse(
                                af.symbol('/')  .andThen(
          function ()   { return term       .andThen(
          function (r)  { return succeed(
             function (x) { return Number(x) / Number(r); }
                                  ); }        ); } )         ).many()
.andThen(
          function (rs) { return succeed(foldl(l, rs));
}                          ); } );

        Eval.sum =               product
.andThen(
          function (l)  { return af.symbol('+')  .andThen(
          function ()   { return product    .andThen(
          function (r)  { return succeed(
             function (x) { return Number(x) + Number(r); }
                                     ); }        ); } ) .orElse(
                                af.symbol('-')  .andThen(
          function ()   { return product    .andThen(
          function (r)  { return succeed(
             function (x) { return Number(x) - Number(r); }
                                  ); }         ); } )        ).many()
.andThen(
          function (rs) { return succeed(foldl(l, rs));
}                          ); } );
```

execute reset

Where appropriate, functions passed to `andThen` have been given a parameter to bind the value produced by the preceding parser and another `andThen` has been added to each sequence (except to the first alternative of `term`) with a function which uses `succeed` to return a value for the recognized expression phrase.

The parsers created using `many` above return lists of curried functions, e.g., the phrase `- 10` will return

```
function (x) { return Number(x) - 10; }
```

and `many` creates a list of such functions. `foldl` is a generally useful class method of `Parser` which takes a value and a (possibly empty) list of curried functions and applies them left to right to accumulate the result value, i.e., `foldl` interprets left associative operation sequences:

`Parser`

```
Parser.foldl = function (l, rs) {
  for (var n = 0; n < rs.length; ++ n)
    l = rs[n](l);
  return l;
};
```

reset

Try this: Input some other expressions.

Try this: Remove `Number` from all functions and evaluate the original input as well as `1+2` and `3*4`. Add code to process the result of `af.number()` so that expressions are again evaluated correctly. ☐

Syntactic sugar

The interpreter for arithmetic expressions shown in the preceding section works, but the implementation is rather error-prone due to the excessive use of nested functions as required by `andThen`. Haskell's *do* notation hugely simplifies specifying computations with monadic values and this section discusses something similar which is needed to make monadic values palatable in JavaScript. The grammar for arithmetic expressions

```
term:    number | '(' sum ')';
product: term ('*' term | '/' term)*;
sum:     product ('+' product | '-' product)*;
expr:    sum eof;
```

can be translated into an interpreter much more literally using a notation for monadic values:

`EvalM`

```
var EvalM = {};  // namespace
with (Parser) with (EvalM) {
  EvalM.af = new Parser.Factory(arithmetic);

  EvalM.term =
    {{{
        af.number();
    |||
        af.symbol('(');
        s <- sum;
        af.symbol(')');
        succeed(s);
    }}};

  term.apply(' 10 ;');
}
```

preprocess

execute reset

`{{{` and `}}}` enclose a computation which will return a monadic value. The computation consists of one or more alternatives, separated by `|||`. An alternative consists of one or more pieces of JavaScript code. Each piece must deliver a monadic value and must be terminated with a semicolon. (Depending on context JavaScript allows a newline to act as a statement terminator but this is not supported here.)

Each piece of JavaScript code may be preceded by an identifier and `<-`. In this case the value *wrapped* by the monadic value produced by the JavaScript code is bound to the identifier, e.g.,

```
x <- succeed(y);
```

will bind the value *y* to the identifier *x*. The scope of the identifier extends from the *next* monadic value to the end of the alternative (i.e., *x* above cannot be used in place of *y* but it can be used beyond `succeed`); the identifier may be shadowed within its scope.

The notation can be nested and suffixes such as `optional`, `many`, and `some` can be applied. This helps to translate the rest of the grammar into monadic notation:

`EvalM`

```
with (Parser) with (EvalM) {
  EvalM.product =
    {{{
        l <- term;
        rs <-
          {{{
              af.symbol('*');
              r <- term;
              succeed(function (x) { return x * r; });
          |||
              af.symbol('/');
              r <- term;
              succeed(function (x) { return x / r; });
          }}}.many();
        succeed(foldl(l, rs));
    }}};

  EvalM.sum =
    {{{
        l <- product;
        rs <-
          {{{
              af.symbol('+');
              r <- product;
              succeed(function (x) { return x + r; });
          |||
              af.symbol('-');
              r <- product;
              succeed(function (x) { return x - r; });
          }}}.many();
        succeed(foldl(l, rs));
    }}};

  EvalM.expr =
    {{{
        s <- sum;
        af.eof();
        succeed(s);
    }}};

  expr.apply(' 10 - 20*30 / (40+50) ');
}
```

preprocess

execute reset

The preprocessor described in a subsequent section converts this notation into the code developed in the previous section.

Try this: The resulting numerical value is not correct. Use the inputs `1+2` and `3*4`, preprocess, execute, and then amend (and preprocess) the first alternative of `term` above to produce the correct

result.

Try this: Add `%` as an operator returning the remainder after division. Add a minus sign operation.

Try this: What happens if there is a division by zero? What happens if a division by zero returns `fail('zero')`? ☐

Some browsers (most notably Apple's Safari) and the JavaScript interpreter *SpiderMonkey* seem to restrict the JavaScript call stack depth and cannot preprocess much larger examples. Fortunately, Firefox and the JavaScript interpreter *Rhino* do not appear to be restricted.

Building trees

It is often convenient to represent input as a tree for further processing. For example, the code from the preceding section can be changed slightly to produce a tree for an arithmetic expression: 

Tree

```
var Tree = { Leaf:0, Add:0, Sub:0, Mul:0, Div:0 };

with (Parser) with (Tree) {
  Parser.makeTreeClasses(Tree);
  Tree.af = new Parser.Factory(arithmetic);

  Tree.term =
    {{{
        n <- af.number();
        succeed(new Tree.Leaf(n));
    |||
        af.symbol('(');
        s <- sum;
        af.symbol(')');
        succeed(s);
    }}};

  term.apply(' 10 ;').value;
}
```

preprocess

execute reset

`term` remains almost unchanged: a *`number`* is represented as a `Leaf` node and whatever `sum` computes is the result of `term`.

Tree

```
with (Parser) with (Tree) {
  Tree.product =
    {{{
        l <- term;
        rs <-
          {{{
              af.symbol('*');
              r <- term;
              succeed(function (x) { return new Tree.Mul(x, r); });
          |||
              af.symbol('/');
              r <- term;
              succeed(function (x) { return new Tree.Div(x, r); });
          }}}.many();
        succeed(foldl(l, rs));
    }}};

  Tree.sum =
    {{{
        l <- product;
        rs <-
          {{{
              af.symbol('+');
              r <- product;
              succeed(function (x) { return new Tree.Add(x, r); });
          |||
              af.symbol('-');
              r <- product;
              succeed(function (x) { return new Tree.Sub(x, r); });
          }}}.many();
        succeed(foldl(l, rs));
    }}};

  Tree.expr =
    {{{
        s <- sum;
        af.eof();
        succeed(s);
    }}};

  expr.apply(' 10 - 20*30 / (40+50) ').value;
}
```

preprocess

execute reset

The functions created in `product` and `sum` are changed to create tree nodes and connect their descendants rather than immediately evaluate the various operations. (The preprocessing area shows the entire tree builder, not just the translation of `expr`, `sum`, and `product`.)

Try this: Add or remove parentheses and otherwise change the arithmetic expression to see how

the tree changes, e.g., to reflect precedence. Do not forget to preprocess whenever the arithmetic expression is changed. □

Tree nodes are so simple that a static factory method `Parser.makeTreeClasses` can arrange for each property of a collection such as `Tree` above to be a JavaScript object constructor; by convention only those properties are modified to be tree classes where the name starts with an upper-case letter:

Parser

```
Parser.makeTreeClasses = function (collection, trace) {
  for (var c in collection)
    if (!c.search(/^[A-Z]/)) {       // change property into tree class
      // constructor
      collection[c] = function () {
        this.content = arguments;
        if (trace) print(this);
      };
      // class name
      collection[c].prototype.className = c;
      // toString
      collection[c].prototype.toString = Parser.dumpTree;
    }
};
```

reset

`makeTreeClasses` installs a constructor which simply saves its `arguments` as `content`. If `makeTreeClasses` is called with a second argument which evaluates to `true`, the constructor will immediately display the new object; this is quite helpful to debug a tree builder.

`makeTreeClasses` stores the `className` as a shared instance variable. This makes it possible to implement a static `dumpTree` function which is connected as `toString` for each of the generated classes:

Parser

```
Parser.dumpTree = function () {
  var indent = '  ' +
      (arguments.length > 0 &&
       typeof arguments[0] == 'string' ? arguments[0] : ''),
    result = this.className + '\n';
  for (var a = 0; a < this.content.length; ++ a)
    if (this.content[a] == null)
      result += indent + 'null\n';
    else if (typeof this.content[a] != 'object')
      result += indent + this.content[a] + '\n';
    else if (this.content[a] instanceof Array) {
      result += indent + '[ ]\n';
      for (var n = 0; n < this.content[a].length; ++ n)
        result += indent+'  ' +
          this.content[a][n].toString(indent+'  ') + '\n';
    } else
      result += indent + this.content[a].toString(indent) + '\n';
  return result.replace(/\n$/, '');
};
```

`dumpTree` returns the class name followed by an indented list of the values in `content`, if any. The list accounts for nested arrays and nested objects, but nested objects are expected to implement `toString` in a compatible fashion.

Try this: Add `%` as an operator returning the remainder after division. Add a minus sign operation.
☐

Converting monadic notation

The monadic notation is embedded in JavaScript and a JavaScript interpreter could be extended to deal with the notation directly. However, especially for prototyping purposes, it is much simpler to convert the notation into JavaScript prior to interpretation. This section discusses the significant parts of a preprocessor implementation; the complete code of the preprocessor is available for editing (and self-preprocessing) on a separate edit page.

The monadic notation can be described by a grammar which concentrates on the monadic computations and obscures most of JavaScript:

```
jsm:      term+ eof;
term:     monad | blanks | word | quoted
    |     '(' term* ')' | '[' term* ']' | '{' term* '}' | symbol;
monad:    '{{{' mvalues ('|||' mvalues)* '}}}';
mvalues:  mvalue+;
mvalue:   blanks? (word blanks? '<-')? term+ ';' blanks?;
```

The grammar shows which parsers (such as *`blanks`*) have to be created with `Parser.Factory` and which literal symbols have to be accepted by one or more of these parsers. Describing the parser factory is the first step to implementing the preprocessor.

`jsm` is JavaScript code and can specify monadic values using `monad` phrases. The preprocessor normalizes white space, i.e., it replaces comments and multiple blanks by single blanks and it preserves all newline characters for the benefit of JavaScript so that there is a trivial correspondence between input and output lines. Therefore the parser factory description does not contain a `skip` property:

```
JSM.scanner = {
  blanks:   /^(\s|\/\/.*|\/\*([^*]|\*+[^\/*])*\*+\/)+/,
  word:     /^[a-zA-Z_][a-zA-Z_0-9]*/,
  quoted:   /^("([^"\\\n]|\\.)*"|'([^'\\\n]|\\.)*'|\(\/([^\/\\\n]|\\.)*\/\))/,
  symbol:   /^(\{\{\{|\|\|\||\}\}\}|<-|[^a-zA-Z_])/,
  eof:      /^$/
};
```

`blanks` describes the white space and comments which have to be normalized for output. `word` describes identifiers to which the monadic notation can bind wrapped values. `quoted` describes strings and regular expressions so that their content cannot be mistaken for symbols. Finally, `symbol` describes the significant multi-character symbols and all single characters which cannot start an identifier -- this does include single digits which make up numbers.

Try this: Add digit sequences as *`number`* to the `scanner` description and to the `term` function

discussed below. □

A reserved list is specified to distinguish symbols which are significant for the grammar from the other symbols which the pattern matches:

```
JSM.scanner.symbol.reserved =
  [ '{{{', '|||', '}}}', '<-', ';', '(', ')', '[', ']', '{', '}' ];
```

Unfortunately, just like strings, regular expressions may contain other characters which then loose their syntactic meaning, but regular expressions are harder to detect than strings because the leading slash may also appear alone as a division operator. To simplify the problem, the preprocessor requires that a regular expression must be enclosed in parentheses without intervening blanks.

The grammar can now be translated into the notation which it describes using methods such as `many`, etc., for the suffixes. For example:

```
with (Parser) with (JSM)
  // mvalue: blanks? (word blanks? '<-')? term+ ';' blanks?
  JSM.mvalue =
    {{{
        pf.blanks().optional();
        {{{
            pf.word();
            pf.blanks().optional();
            pf.symbol('<-');
        }}}.optional();
        term(true).some();
        pf.symbol(';');
        pf.blanks().optional();
    }}};
```

Once a preprocessor is available, this code can be converted to JavaScript and executed to check if input conforms to the grammar. Unfortunately, for the initial version of the preprocessor this code had to be hand-translated using `andThen`, etc., in the style shown before.

The code fragment hints at a complication: `term` is used three times in the grammar: at the level of JavaScript code in `jsm`, at the level of monadic code in `mvalue` as shown above, and recursively, enclosed by various parentheses in `term` itself:

```
term: monad | blanks | word | quoted
    | '(' term* ')' | '[' term* ']' | '{' term* '}' | symbol;
```

In `mvalue` a semicolon is significant but elsewhere it is not, i.e., the wildcard *`symbol`* cannot always match a semicolon. Fortunately, `term` is encoded as a monadic value, i.e., as a data item which can be created by a function with a parameter which controls whether or not a semicolon should be recognized when the data item is applied:

```
with (Parser) with (JSM)
  JSM.term = function (noSemicolon) {
    return (
      {{{
          monad;
      |||
```

```
            pf.blanks();
        |||
            // ...
        |||
            pf.symbol(';');
            noSemicolon ? fail('semicolon not expected')
                        : succeed(';');
        |||
            pf.symbol('(');
            term(false).many();
            pf.symbol(')');
        |||
            // ...
        |||
            pf.symbol(); // all non-reserved symbols
        }}}};
  };
```

JavaScript's handling of newlines can result in ambiguities -- this is why `blanks` are explicit in this grammar so that newlines can be passed from monadic notation to preprocessed JavaScript. Specifically, if `return` and an argument are separated by a newline, JavaScript will treat `return` alone as a statement! As the code above shows this is easily circumvented by enclosing a `return` argument in parentheses and inserting a newline, if any, only *after* the leading parenthesis.

Once a grammar has been translated into monadic notation -- and executed to recognize some input if at all possible -- it needs to be extended with code which will represent or interpret the input. The preprocessor uses the factory method `makeTreeClasses` discussed in the previous section; the problem of translating the input is thus delegated to adding appropriate methods to these classes later. The necessary classes can be discovered top-down, simply by adding identifiers to bind interesting values returned by parsers and adding constructor calls in `succeed` calls to represent the interesting values. The monadic notation is ideally suited to this approach:

```
with (Parser) with (JSM)
  JSM.term = function (noSemicolon) {
    return (
      {{{
          monad;
      |||
          b <- pf.blanks();
          succeed(new Blank(b));
      |||
          // ...
      |||
          pf.symbol(';');
          noSemicolon ? fail('semicolon not expected')
                      : succeed(new Text(';'));
      |||
                pf.symbol('(');
          ts <- term(false).many();
                pf.symbol(')');
          succeed(new Paren('(', ts, ')'));
      |||
          // ...
      |||
          s <- pf.symbol(); // all non-reserved symbols
          succeed(new Text(s));
```

```
            }}});
        };
```

`Paren` and `Text` can be implemented using `makeTreeClasses` but `Blank` is best coded manually because white space is normalized before it is passed through:

```
JSM.Blank = function (value) {
  this.value = value.replace(/./g, '');  // remove all but newlines
  if (!this.value) this.value = ' ';     // if no newlines: single blank
  if (trace) print(this);
};

JSM.Blank.prototype.toString = function () {
  switch (this.value) {
  case ' ':  return 'blank';
  case '\n': return 'newline';
  default:   return this.value.length+' newlines';
  }
};
```

The grammar and the monadic notation can be changed to perform some input rewriting or validation which cannot be conveniently expressed in the grammar itself. For example, the original rule

```
mvalue:  blanks? (word blanks? '<-')? term+ ';' blanks?;
```

can be rewritten as

```
mvalue:  ( blanks word blanks? '<-'
         |         word blanks? '<-'
         | blanks?                 )  term+ ';' blanks?;
```

because this makes it simple to combine the first two optional pieces of white space before handing them to the constructor. Moreover, `term` can be *`blanks`* and this is reasonable wherever `term` is used in this grammar -- empty parentheses or braces, no JavaScript code at all, etc. -- except in the situation shown above: a monadic value cannot be empty. This needs to be checked before `mvalue` is accepted:

```
with (Parser) with (JSM)
  JSM.mvalue =
    {{{
        bw <- {{{
                  b1 <- pf.blanks();
                  w  <- pf.word();
                  b2 <- pf.blanks().optional(null);
                  pf.symbol('<-');
                  succeed([new Blank(b1 + (b2 ? b2 : '')), w]);
              |||
                  w  <- pf.word();
                  b2 <- pf.blanks().optional(null);
                  pf.symbol('<-');
                  succeed([b2 ? new Blank(b2) : null, w]);
              |||
                  b1 <- pf.blanks().optional(null);
                  succeed([b1 ? new Blank(b1) : null, null]);
              }}};
```

```
        ts <- term(true).some();
        pf.symbol(';');
        b3 <- pf.blanks().optional(null);
        (function () {
          for (var n = 0; n < ts.length; ++ n)
            if (!(ts[n] instanceof Blank))
              return succeed(new Mvalue(bw[0], bw[1], ts,
                                b3 ? new Blank(b3) : null));
          return fail('expecting monadic value');
        })();
      }}};
```

`bw` is bound to a list containing `null` or `Blank` for the first two pieces of white space and `null` or a string for the identifier; `ts` is bound to a list of terms which will not include semicolons; finally, `b3` is bound to `null` or a string with the last piece of white space, if any. All that is left is a loop over the `term` list to see if it contains a non-`Blank` and if so, the `mvalue` parser succeeds and the `Mvalue` constructor can be called with a canonical representation of the possible descendants.

Programming languages often make a difference between statements and expressions. The monadic notation requires that each computation produce a monadic value so that it can be unwrapped and bound to an identifier if one is specified. This means that a computation has to be a JavaScript expression, i.e., it can use conditional evaluation with `? :` but it cannot use a loop. However, the code above shows that one can always insert a parameterless, anonymous function containing statements and call the function immediately following the definition. Among the statements must be at least one `return` to create the required value.

Finally, given a tree representing a JavaScript program with embedded `monad` phrases, code generation is implemented as a method `gen` in each of the classes from which the tree is built:

```
JSM.Blank.prototype.gen = function () {
  return this.value;
};

JSM.Paren.prototype.gen = function () {
  var content = this.content[1],
    result = '';
  for (var n = 0; n < content.length; ++ n)
    result += content[n].gen();
  return this.content[0] + result + this.content[2];
};
```

`Text` and `Blank` simply emit text or normalized white space, respectively. `Paren` emits the delimiters and uses a loop to generate code for the `content` between the delimiters. The hard part of the conversion is accomplished by `Monad`, `Mvalues`, and `Mvalue`. A `Monad` tree node contains a non-empty list of `Mvalues` and uses a loop to connect their code with `orElse` if needed:

```
// Monad: Mvalues+
//   Mvalues .orElse( Mvalues ) ...
JSM.Monad.prototype.gen = function () {
  var content = this.content[0],
    result = content[0].gen();
  for (var n = 1; n < content.length; ++ n)
    result += '.orElse(' + content[n].gen() + ')';
  return result;
};
```

`Mvalues` contains a non-empty list of `Mvalue` objects and starts a loop implemented by recursion so that they can implement the appropriate function parameter name and connect their code with `andThen` if needed:

```
// Mvalues: Mvalue+
JSM.Mvalues.prototype.gen = function () {
  var content = this.content[0];
  return content[0].gen('', content, 1);
};

// Mvalue: Blank? word? (Blank|Monad|Paren|Text)+ Blank?
//   Blank? term+ Blank? .andThen(function (word?) { return ... })?
Mvalue.prototype.gen = function (head, content, next) {
  if (this.content[0]) head += this.content[0].gen();
  for (var n = 0; n < this.content[2].length; ++ n)
    head += this.content[2][n].gen();
  if (this.content[3]) head += this.content[3].gen();
  if (next < content.length) {
    head += '.andThen(function (';
    if (this.content[1]) head += this.content[1];
    head += ') { return (';
    head = content[next].gen(head, content, next+1);
    head += '); })'
  }
  return head;
};
```

Normally the constructor calls are issued by the `succeed` clauses in the monadic notation; however, for testing purposes a tree can be built, displayed, and converted interactively:

JSM_Tree

```
with (Parser) with (JSM) {
  new Monad([
      new Mvalues([
          new Mvalue(null, 'n', [
              new Text('number')
            ], null),
          new Mvalue(null, null, [
              new Text('succeed'),
              new Paren('(', [new Text('n-0')], ')')
            ], null)
        ])
    ]) // .gen()
}
```

execute reset

Try this: Apply `gen` as indicated to see the generated code. ☐

But for white space, this example generates the same code as the monadic notation:

JSM_Tree

```
{{{
    n <- number;
    succeed(n-0);
}}}
```

preprocess reset

The tree display is always very useful to design code generation methods. It is instructive to take the tree output above and walk through the `gen` methods for the various preprocessor tree classes.

A monadic interpreter

This section discusses the implementation of a little imperative programming language. The implementation uses monadic classes and consists of a parser to build a tree to represent the target program and of an interpreter which evaluates the tree. The complete code of the system is available for editing and preprocessing on a separate edit page.

The following program implements Euclid's algorithm to compute the greatest common divisor of two natural numbers. It uses typical features of such a little imperative programming language:

Mini.interpret()

```
// greatest common divisor

{ x = 36
  y = 54
  while x <> y do
    if x > y then
      x = x - y
    else
      y = y - x
  print x
}
```

interpret reset

The following function controls the processing of a source string written in the little language:

Mini

```
with (Mini) with (Memory) {

  Mini.interpret = function (source) {
    // compile
    var tree = prog.apply(source);
    if ('fail' in tree)
      print(tree.fail);

    else {
      // uncomment to see the tree
      // print(tree.value);

      // make interpreter
      var monad = tree.value.eval();

      // uncomment to see the interpreter
      // print(dump(monad));

      // execute, produce final environment
      // try new ClonedHash()
      return monad.apply(new Hash());
    }
  };
}
```

reset

Try this: Change the little language example above to obtain the greatest common divisor of other pairs of natural numbers.

Try this: Introduce an error, e.g., add a semicolon after `36`.

Try this: Extend the example to compute the least common multiple of the two numbers. □

The parser and tree builder for the little language is implemented just like the preprocessor discussed in the previous section. The grammar is an extension of the grammar for arithmetic expressions presented earlier:

```
term:       number | word | '(' sum ')';
product:    term ('*' term | '/' term)*;
sum:        product ('+' product | '-' product)*;
comparison: sum ('<' sum | '<=' sum
          | '>' sum | '>=' sum | '<>' sum | '=' sum);
stmt:       sum | word '=' sum | 'print' sum
    |       'if' comparison 'then' stmt ('else' stmt)?
    |       'while' comparison 'do' stmt
    |       '{' stmt* '}'
    |       'try' stmt 'catch' (word ':')? stmt
    |       'raise' quoted;
prog:       stmt eof;
```

The grammar can be translated into monadic notation as soon as the collection of regular expressions for the `Parser.Factory` has been defined:

```
Mini.scanner = {
  skip:    /^(\s|\/\/.*|\/\*([^*]|\*+[^\/*])*\*+\/)+/,  // Java comments
  word:    /^[a-zA-Z_][a-zA-Z_0-9]*/,                   // identifiers
```

```
  number:  /^[0-9]+/,                                   // integers
  quoted:  /^'[^'\n]*'/,                                // simple strings
  symbol:  /^(<=|>=|<>|.)/,
  eof:     /^$/
};
Mini.scanner.word.reserved = [
  'catch', 'do', 'else', 'if', 'print', 'raise', 'then', 'try', 'while'
];
```

Once the grammar is translated, it is easy to see which classes are needed to represent a program. They can all be generated using `makeTreeClasses`:

```
var Mini = {
  // arithmetic
  Add: 1, Div: 1, Mul: 1, Sub: 1,           // left right
  Leaf: 1,                                  // number
  Name: 1,                                  // string
  // comparisons
  Eq: 1, Ge: 1, Gt: 1, Le: 1, Lt: 1, Ne: 1,  // left right
  // statements
  Assign: 1,                                // string sum
  Block: 1,                                 // stmt+
  Expr: 1,                                  // sum
  If: 1,                                    // comparison stmt stmt?
  Print: 1,                                 // sum
  While: 1,                                 // comparison stmt
  // exception handling
  Raise: 1,                                 // string
  Try: 1                                    // stmt name? stmt
};
```

Try this: Change `interpret` above to display the tree which represents Euclid's algorithm. Execute the example and compare the tree to the `Mini` collection. Dump the interpreter.

Try this: On the edit page, change `scanner` and `stmt` so that a block must be enclosed with words such as `begin` and `end`. Change the example and test.

Try this: Test nested `if` statements. What happens to a "dangling" `else` clause? Use the edit page for `Mini` and make `else` mandatory. □

The interpreter for the little language can be implemented with the *visitor* design pattern or by adding `eval` methods to the tree classes. An interpreter maps the features of the target language to the host language and it has to simulate missing features; e.g., Haskell does not support assignment and exception handling, i.e., if an interpreter is written in Haskell those features have to be simulated using monadic values.

JavaScript supports both, global assignment and exception handling, and there is no obvious need for simulation. However, monadic values make it possible to separate host and target behavior. The interpreter discussed here uses a monadic class `Memory` which holds an environment of current variable names and values accessible through `fetch` and `store` methods as the evaluation proceeds. The environment can be implemented as a collection:

```
// collection-based environment.
Mini.Hash = function () { };

Mini.Hash.prototype.fetch = function (name) {
```

```
  return this[name];
};

Mini.Hash.prototype.store = function (name, value) {
  this[name] = value;
  return this;
};
```

All evaluation methods must return monadic values, for example:

```
// monad with an environment
Mini.Memory = Monad.subclass();

with (Mini) with (Memory)
  Leaf.prototype.eval = function () {
    var value = this.content[0];  // number
    return succeed(Number(value));
  };
```

Evaluation methods for binary operators can be implemented with a factory function `bin` which receives the actual operation as a parameter and creates the evaluation method with monadic notation, for example:

```
Mini.bin = function (op) {       // operation as a function
  return function () {
    var left = this.content[0],  // left operand tree
      right = this.content[1];   // right operand tree
    return (
      {{{
          l <- left.eval();
          r <- right.eval();
          succeed(op(l, r));
      }}});
  };
};

with (Mini) with (Memory)
  Add.prototype.eval = bin(function (l, r) { return l + r; });
```

`Memory` maintains an environment with `fetch` and `store` operations as state. Access to the state is combined with these operations to implement variable reference and assignment:

```
with (Mini) with (Memory) {
  Name.prototype.eval = function () {
    var name = this.content[0];  // variable name
    return (
      {{{
          env <- get;
          succeed(env.fetch(name));
      }}});
  };

  Assign.prototype.eval = function () {
    var name = this.content[0],  // variable name
      sum = this.content[1];     // expression tree
    return (
      {{{
          value <- sum.eval();
```

```
            env <- get;
            put(undefined, env.store(name, value));
        }}});
  };
}
```

Once monadic values are employed, `andThen` must be used to sequence execution at the statement level. *if* can be mapped to conditional evaluation; a missing *else* part has to be fudged:

```
with (Mini) with (Memory)
  If.prototype.eval = function () {
    var c = this.content[0],  // condition tree
      t = this.content[1],    // then tree
      e = this.content[2];    // else tree
    return (
      {{{
          cond <- c.eval();
          cond ? t.eval() : (e ? e.eval() : succeed(undefined));
      }}});
  };
```

A sequence of statements requires a loop which in turn requires a recursive implementation:

```
with (Mini) with (Memory)
  Block.prototype.eval = function (_n) {
    var self = this,              // for closure
      n = _n ? _n : 0,            // for tail recursion
      content = this.content[0];  // the statement list
    if (n >= content.length)
      return succeed(undefined);  // completed block has no value
    else
      return (
        {{{
            content[n].eval();    // sequencing must be by andThen
            self.eval(n+1);
        }}});
  };
```

Similarly, *while* must also be based on recursion:

```
with (Mini) with (Memory)
  While.prototype.eval = function () {
    var self = this,              // for closure
      c = this.content[0],        // condition tree
      body = this.content[1];     // body tree
    return (
      {{{
          cc <- c.eval();
          (function () {
            if (!cc) return succeed(undefined);
            return (
              {{{
                  body.eval();
                  self.eval();
              }}});
          })();
      }}});
  };
```

For a host language like JavaScript, which has all the features of the target language, this approach looks too complicated. However, a monadic class can be used to implement exception handling very elegantly and with a choice of semantics:

Mini.interpret()

```
// division by zero

// try
  1/0
// catch 2
// catch e: print e
```

interpret reset

Try this: Uncomment `try` and the first `catch`.

Try this: Uncomment `try` and the second `catch`. ☐

Exception handling is based on `fail` and `orElse`, i.e., an operation such as division can use `fail` and cause an exception which a little language program can catch:

```
with (Mini) with (Memory)
  Div.prototype.eval = function () {
    var left = this.content[0],  // left operand tree
      right = this.content[1];   // right operand tree
    return (
      {{{
          l <- left.eval();
          r <- right.eval();
          r ? succeed(l / r) : fail('division by zero');
      }}});
    };
```

A failure can be caught with `orElse`, i.e., with `|||` in monadic notation:

```
with (Mini) with (Memory)
  Try.prototype.eval = function () {
    var a = this.content[0],  // try body tree
      n = this.content[1],    // catch variable name if any
      b = this.content[2];    // catch body tree
    if (!n)  // no catch variable name
      return (
        {{{
            a.eval();
        |||
            b.eval();
        }}});
```

The `try` body `a` is evaluated. If there is a failure, the `catch` body `b` is evaluated. Both use the same initial state (environment).

It is even possible to make the error message from `fail` available as a variable value in the catch body. This is accomplished by re-implementing with a function argument in the style of `andThen`:

```
function onError (a, b) { // re-implement orElse with a parameter
  return new Memory(
    function (state) {
      var result = a.apply(state);
      return 'fail' in result ? b(result.fail).apply(state) : result;
    }
  );
}
```

The alternative `b` must be wrapped as a function in order to provide the parameter scope for whatever value should be passed into the alternative. With `onError`, exception handling can be implemented so that the exception handler has access to the failure message:

```
return onError(a.eval(),
  function (value) {
    return (
      {{{
          env <- get;
          put(undefined, env.store(n, value));
          b.eval();
      }}});
  });
};
```

The `try` body `a` is evaluated unconditionally. If it fails, the failure message is bound to the parameter `value` and from there to the variable name `n` in the environment before the `catch` body `b` is evaluated, i.e., `b` has access to the failure message with the variable name `n`.

Perhaps the most interesting aspect of this implementation of exception handling is that the interaction between assignment and exception handling can be influenced by the implementation of the environment.

Mini.interpret()

```
// exception handling

try {
  foo = 10
  bar = 20
  raise 'error'
} catch
  foobar = 30
```

interpret reset

This example produces a final environment which contains values for `foo`, `bar`, *and* `foobar`, i.e., `catch` seems to have operated on the state at the point of `raise` and not -- as should be the case for a monad-based implementation -- on the state at `try`. This behavior is caused by `Hash`, the implementation of the environment: As shown above, `catch` is implemented using `orElse`, i.e., `try` and `catch` do start out with the same environment. However, `Hash` is a simple JavaScript collection, i.e., all changes are persistent. The persistence can be removed, e.g., by cloning the environment whenever it is changed (there are less expensive ways):

```
Mini.ClonedHash = function () { };

Mini.ClonedHash.prototype.fetch = function (name) {
  return this[name];
};

Mini.ClonedHash.prototype.store = function (name, value) {
  var result = new Mini.ClonedHash();
  for (var n in this)
    if (this.hasOwnProperty(n))
      result[n] = this[n];
  result[name] = value;
  return result;
};
```

Try this: Change `interpret` near the beginning of this section to use a `ClonedHash` environment and execute the example again. □

If the environment is not persistent, the assignments in the `try` block are undone if there is an exception. While the implementation is expensive, the fact that this happens invisibly might make this variant of a `try` statement quite attractive, e.g., in a backtracking context.

Summary

This web page discusses a functional programming style suggested by the *Monad* and *MonadPlus* classes of Haskell and introduced a notation to facilitate coding in JavaScript. The significant problem domain is parsing: support for monadic parsers exists for Haskell , Python , and other languages .

This web page describes monadic LL(n) parsing with JavaScript. It contains the implementation of `Monad`, a base class for monadic JavaScript classes which wrap state functions. Subclasses can be created with the class method . A subclass has the shared monadic value to fetch the current state; the class methods to create a monadic value to store a new value and state, and to create monadic values reporting success and failure, and to serialize collections; and the methods to apply a monadic value, and and to create monadic values which apply monadic values sequentially or as alternatives.

This web page contains the implementation of `Parser`, a `Monad` subclass of monadic parsers which can be combined to represent LL(n) grammars. `Parser` provides the additional combinators , , and to support representing EBNF grammars. `Parser` has a number of class methods. supports left-associative assembly of a list of functions and converts certain property names in a collection into classes to represent trees which are connected to for serialization. is a class which is constructed with a collection of regular expressions and has methods to construct parsers which accept input matching the regular expressions.

This web page contains a preprocessor which extends JavaScript with a which simplifies specifying computations which involve monadic values. In particular, the notation facilitates translating LL(n) grammars into combinations of `Parser` values. From this perspective, `JSM` is a parser generator for JavaScript and the monadic notation extends JavaScript as input language for the parser generator. The web page contains three larger examples which use the preprocessor: a tree builder for a small programming language, an interpreter for the trees which implements error recovery and a

functional approach to assignment, and finally the preprocessor itself.

`Monad` only requires functions as first-order values, `Parser` additionally benefits from regular expressions. Dynamic typing is convenient but not essential. Therefore, this web page can serve as a blueprint for quickly implementing parser generators in other programming languages where these features are available or can be simulated.

Finally, all code in this web page can be edited and used interactively. Therefore, the support files of this web page can be used as a framework for interactive, literate programming in JavaScript.

Download Files

code/getStdin.js	an attempt at reading source from standard input.
code/jsm.js	command line program to run the preprocessor
code/mini.js	command line program to compile and execute the little language
paper/	infrastructure for interactive, literate JavaScript tutorials.

Download Namespaces

`arithmetic`	patterns for arithmetic expressions.
`Axioms`	examples illustrating the monad axioms.
`Eval`	evaluate arithmetic expressions.
`EvalM`	evaluate arithmetic expressions (monadic notation).
`Expr`	recognize arithmetic expressions.
`Expr_badSum`	recognize arithmetic expressions (defective).
`JSM`	preprocessor to convert monadic notation to JavaScript.
`JSM_Tree`	illustrate tree for mandic notation.
`language`	patterns for a small programming language.
`Mini`	interpreter for a small programming language.
`Monad`	abstract base class for monadic classes.
`Parser`	monadic parser.
`Scanners`	examples illustrating parsing.
`Tree`	represent arithmetic expressions as trees.
changes	create a page with all current changes to the original code.

References

[1] *Haskell - HaskellWiki*, `http://haskell.org/` retrieved on June 30, 2008.

[2] *The C# Language*, `http://msdn.microsoft.com/en-us/vcsharp/aa336809.aspx` retrieved on June 30, 2008.

[3] *Groovy - Home*, `http://groovy.codehaus.org/` retrieved on June 30, 2008.

[4] *Closures for the Java Programming Language*, `http://javac.info/` retrieved on June 30, 2008.

[5] *ECMAScript Language Specification*, Edition 3, http://www.mozilla.org/js/language/E262-3.pdf retrieved on June 30, 2008.

[6] Guido van Rossum, *Python Reference Manual*, http://docs.python.org/ref/ retrieved on June 30, 2008.

[7] *Ruby Home Page*, http://www2.ruby-lang.org/en/ retrieved on June 30, 2008.

[8] *The Scheme Programming Language*, http://www-swiss.ai.mit.edu/projects/scheme/ retrieved on June 30, 2008.

[9] *Rhino - JavaScript for Java*, http://www.mozilla.org/rhino retrieved on June 30, 2008.

[10] *SpiderMonkey (JavaScript-C) Engine*, http://www.mozilla.org/js/spidermonkey retrieved on June 30, 2008.

[11] Graham Hutton, *Programming in Haskell*, Cambridge University Press, 2007.

[12] Niklaus Wirth, *The Programming Language Pascal* (Revised Report), http://www.standardpascal.org/The Programming Language Pascal 1973.pdf retrieved on June 30, 2008.

[13] *RFC-Editor Webpage*, http://www.rfc-editor.org/ retrieved on June 30, 2008.

[14] *Firefox web browser*, http://www.mozilla.com/firefox/ retrieved on June 30, 2008.

[15] Daan Leijen, *Parsec, a fast combinator parser*, http://research.microsoft.com/users/daan/download/parsec/parsec.pdf retrieved on June 30, 2008.

[16] *Pysec: Monadic Combinatoric Parsing in Python*, http://www.valuedlessons.com/2008/02/pysec-monadic-combinatoric-parsing-in.html retrieved on June 30, 2008. April 1, 2008

[17] Monad (functional programming), *Wikipedia*, http://en.wikipedia.org/wiki/Monads_in_functional_programming#External_links retrieved on June 30, 2008.