2008

Functional Parsing — A Multi-Lingual Killer-Application

Axel-Tobias Schreiner
Rochester Institute of Technology

James Heliotis
Rochester Institute of Technology

Follow this and additional works at: http://scholarworks.rit.edu/other

Recommended Citation

This Presentation is brought to you for free and open access by RIT Scholar Works. It has been accepted for inclusion in Presentations and other scholarship by an authorized administrator of RIT Scholar Works. For more information, please contact ritscholarworks@rit.edu.
ABSTRACT
Monads are used in Haskell to support error handling and a concept of global state such as input/output. Parsing is a killer application for monads. This paper discusses a parser generator implementation for Java and JavaScript and shows how to benefit greatly from object-oriented design patterns based on monads adapted from functional programming.

All examples discussed here are available for online use through the links at the end of the paper.

1. FUNCTIONAL PARSING
A parser function accepts input and produces a result that either indicates failure or consists of a value indicating what was recognized and the rest of the input. The recognized value might be a string or number representing an input symbol, a collection of nodes representing structured input, or something entirely different.

If functions are first-order values parser can be a type where each value is a parser function. In Java we must create an interface to hold the function à la Command pattern:

```java
public interface ParserFunction {
    Result f (Input i);
}
```

Parser then becomes a class with a constructor that accepts a ParserFunction. This is a variation on the State pattern in that the variable behavior of Parser objects is placed outside of them instead of inside by using (static) subclasses. The classes Input and Result can be considered abstract for now.

A Java class needs methods. The core Parser method is parse: it accepts Input and returns the Result from executing its object’s ParserFunction.

In Haskell [1] a data type can be an instance of a class and the class declares methods that the data type must implement. Haskell can view the parser function as a state function: it accepts a global state (input) and returns a value and another state (remaining input). State functions can be values of a data type that can be an instance of the classes Monad and MonadPlus [2]. These classes require the data type to implement certain methods which prove to be very useful in the context of parsing. This leads to a number of additional Parser methods.

succeed (called return in Haskell) is a method that accepts a value and returns a Parser with a ParserFunction that accepts any input and returns a result with the specified value and the unchanged input. It provides a way for the overall parsing process to be “happy” with the current state (input) and to provide an explicit value as result of the ParserFunction. (See optional and many below.) In Java, succeed can be implemented as a static method of the Parser class and values are represented as Object.

fail is a method that accepts an error message and returns a Parser with a ParserFunction that will always return a result indicating failure. It provides a way to force the overall parsing process to fail, independent of the current state (input). In Java, fail can be implemented as a static method of the Parser class.

Beyond the above primitive parsers, we can combine parsers to wrap algorithms that utilize one or more simpler parsers to create their behavior. This is in the spirit of the Template pattern. There is a strong correspondence between the following methods and common grammatical operators.

orElse (mplus in Haskell) is used for expressing alternatives. It accepts two parsers and builds and returns a new parser with a parser function that executes the first and, in case of failure, the second parser function, either one with the same input. In Java, orElse can be implemented as a method of the Parser class where the first parser is the receiver and the second parser is the argument. (Some local variables are marked final to allow them to be accessed by nested classes, i.e., to achieve closure.)

```java
public Parser orElse (final Parser b) {
    final Parser a = this;
    return new Parser() {
        public Result f (Input i) {
            Result result = a.parse(i);
            return result.failed ? b.parse(i) : result;
        }
    };
}
```

The grammatical expression a | b will manifest itself in Java code as a.orElse(b).
Similarly, andThen (referred to in Haskell as the bind operation) combines two parsers for sequential execution and, in case of success, allows access to the value recognized by the first parser function. andThen creates a parser with a parser function which applies the first parser’s function to the input. If it fails, that failing result is returned. Otherwise andThen’s parser function runs the second parser’s function on the remaining input and returns that result as its result.

There is a hitch; however. The second parser is constructed based on the value generated by the first parser function. This is an example of dynamic generation of a parser function based on run-time context. We call this context Scope. Instead of a Parser instance as the second argument to andThen, a Scope instance, which can generate a Parser given the first parse’s result value, is provided as the second argument. In Java, Scope is fairly simple:

```java
public interface Scope {
  Parser s (Object firstValue);
}
```

andThen is a Parser method where the first argument (the parser) is the receiver and the second argument, the Scope, is the method’s sole argument:

```java
public Parser andThen (final Scope b) {
  final Parser a = this;
  return new Parser(new ParserFunction() {
    public Result f (Input i) {
      Result result = a.parse(i);
      return result.failed ? result :
        b.s(result.value).parse(result.input);
    }
  });
}
```

andThen is akin to the grammatical construct of concatenation, i.e., a b, together with an instance of Scope.

We now have the basic building blocks from Haskell in place: success, failure, alternation, concatenation, and also recursion, a side-effect of using Java functions. However, three more Parser combinators should be implemented to simplify expressing optional elements and repetition in a grammar. These methods also serve as examples for the use of the fundamental combinators introduced thus far.

optional uses a parser and a value and returns a parser with a parser function which will return the result of the given parser function on success or the value and the original input on failure. In Java, optional can be implemented as a Parser method where the parser is the receiver and the value is the argument:

```java
public Parser optional (Object value) {
  return this.orElse(succeed(value));
}
```

optional is akin to the suffix ? which indicates an optional phrase in EBNF (in the style used by the Internet Request For Comments [3]).

Similarly, some is applied to a parser that we will call self. It returns a parser with a parser function that will apply self’s parser function one or more times and return a result with a non-empty list and the ultimately remaining input. The list contains, in order, all the values returned in the results of the successive calls to self’s parser function. In Java, some can be implemented as a Parser method where the parser is the receiver:

```java
public Parser some () {
  final Parser self = this; // for closure
  return self.andThen(new Scope() {
    public Parser s (final Object fromSelf) {
      return self.many().andThen(new Scope() {
        public Parser s (final Object fromMany) {
          ArrayList result = new ArrayList();
          result.addAll((ArrayList)fromMany);
          result.add(fromSelf);
          return succeed(result);
        }
      });
    }
  });
}
```

some is akin to the suffix + which indicates one or more iterations of a phrase in EBNF. x+ can be expressed as x x* and the implementation uses andThen for concatenation and succeed to return the list of values. The suffix * corresponds to the method many, which is covered next.

many is a combination of some and optional. It returns a parser with a parser function that will act like some if successful and like optional if not. More precisely, just like some it will return a list of values and the ultimately remaining input. If some does not succeed at all, the result of many contains an empty list and the original input. In Java, many can be implemented as a Parser method where the parser is the receiver:

```java
public Parser many () {
  return this.some().optional(new ArrayList());
}
```

All of these methods together represent exactly the operations that are permitted in the EBNF grammar notation. If a grammar is expressed in EBNF, longest alternative first, and is not left-recursive, it can be literally translated into calls on these methods and the result will be a parser implementing the grammar. As an example here is the typical alphanumeric identifier:

```batch
id: letter { letter | digit }*
```

Given the parsers letter and digit, this grammar would translate into
Parser id = letter.andThen(
  new Scope() { public Parser s (final Object fromLetter) {
    return letter.orElse(digit).many().andThen(
      new Scope() { public Parser s (final Object fromMany) {
        String result = fromLetter.toString();
        for (Object o : (ArrayList)fromMany) result += o;
        return succeed(result);
      }
    });
  }
});

For small demonstration exercises the work presented so far may suffice. However, for more significant projects two problems remain: writing parsers economically for terminal symbols such as letter and digit (or even identifier), and a more reasonable notation in which to represent the grammar than the pure Java code used so far.

2. FUNCTIONAL SCANNERS

Hutton [4] shows how to create letter, digit, and similar parsers just from character classification predicates, but numerous languages implement regular expressions, often called patterns, which are more concise and more efficient for character-oriented parsing. A ParserFactory is constructed with patterns describing insignificant input such as white space or comments, strings to be matched literally such as operators or keywords, and patterns describing sets of input symbols such as identifiers or numbers:

ParserFactory pf = new ParserFactory(
  new String[]{ "\s" }, // insignificant: space
  new String[]{ "(" }, // literals: ( )
  new String[]{ "[0-9]+" } // patterns: number
);

ParserFactory is derived from a new class Lexer that uses patterns to classify input, quite like Java’s StreamTokenizer uses character classes. The parsers returned by a ParserFactory contain parser functions that are all based on a Lexer method accept that accepts an Input value and returns another Input value for the remaining input; accessor methods such as token and value provide information about what was accepted.Lexer should be suitable to solve more general text partitioning problems, e.g., for scanners for the parser generators jay [5] or oops3 [6].

Input simply encapsulates an input position in some underlying store. Different implementations support strings or input streams as stores, and mutable or immutable input values. Therefore, Lexer can be used for interactive input as well as functional parsing.

In the example presented later in the paper the methods literal, pattern, and eof of ParserFactory return parsers with parser functions that succeed if the significant input is a literal string, if the input matches a pattern (selected by position in the list used for construction), or if there is no more significant input, respectively.

3. A MONADIC NOTATION

Between ParserFactory and Parser it is now quite simple to translate an EBNF grammar and patterns describing tokens such as id into a Parser that will recognize sentences conforming to the grammar. The implementation of id above shows how to add code to construct a representation of the sentence. It also shows, however, that a larger example produces pretty ugly code. The typical term in an arithmetic expression

term: number | '(' expression ')';

using the ParserFactory defined in the previous section is translated into

Parser term () {
  return {{
    pf.pattern(0).orElse( // [0-9]>*
      literal("(").andThen( // [0-9]>*
        new Scope() { public Parser s (Object ignored) { return expression().andThen( // [0-9]>*
          new Scope() { public Parser s (final Object e) { return literal(")");.andThen( // [0-9]>*
            new Scope() { public Parser s (Object ignored) { return succeed(e); }
          }
        }
      }
    });
  };
  });
}

Haskell’s do notation makes it clear that computing with monadic values can be greatly simplified by a more suitable syntax. Assuming that {{ and }} enclose a monadic value, i.e., a Parser, and that ||| represents orElse, the example above becomes

Parser term () {
  return {{
    pf.pattern(0);
    ||| literal("(");
    e <- expression();
    literal(")");
    succeed(e);
  }};
}

A semicolon is used to terminate a monadic value, i.e., it indicates sequential execution. A binding consisting of an identifier followed by <- can precede a monadic value; its scope extends to the end of the sequence. (This is why Scope was chosen as the name for the contextual class introduced above.)
The notation is so loosely coupled with the host language that a preprocessor based on the following grammar can implement it:

```
program: code+ eof;

code: monad | blanks | word | string
    | '{' code* '}'
    | '[' code* ']
    | '{' code* '}' | symbol;
monad: '{{' value+ ('|||' value+)* '}}';
value: blanks? (word blanks? '<-')?
    | code+ ';' blanks?;''
```

The preprocessor preserves white space so that input lines correspond exactly to output lines. Its `ParserFactory` has patterns for `blanks` describing white space and comments, `word` describing Java identifiers, `string` describing character and string literals, and `symbol` describing all single characters not mentioned explicitly in the grammar above.

`program` describes a complete program in the host language which may contain monad phrases. These contain "some" sequential `value` phrases, each optionally including a binding, and "many" instances of the operator `|||` to represent `orElse`. In reality there are two definitions for `code`: within `program` a semicolon is simply a `code`, but nested within `monad` a semicolon is significant as part of a `value`.

A tree of nodes derived from `AbstractList` can represent recognized input. The subclasses can be generated by the preprocessor once their names are known; therefore, `nodes` is added as yet another alternative of `code`:

```
nodes: '{{' blanks? (word blanks?+ '})});
```

Distinguishing `nodes` and `monad` requires arbitrary lookahead — `nodes` does not contain a semicolon. Unlike other parser generators the functional parsing algorithm is powerful enough to deal with this complication.

Given the preprocessor, a grammar such as the one for `term` is translated into monadic notation and preprocessed. A `ParserFactory` is constructed to provide simple parsers for the terminal symbols of the grammar. Code with `succeed` is added to represent input as a tree or to perform evaluation, etc. Alas, while the preprocessor can translate itself, there is always a first time...

4. REVIEW

The paper demonstrates that one language can well benefit from emulating features of another, even if there is a gross mismatch in programming paradigms. This project started with the question what it really takes to create monadic values — do functions as first-order values suffice? Mastery of monads is fundamental to using Haskell well but the concept is quite confusing to the beginning Haskell programmer [1]. JavaScript was initially chosen for an experimental implementation in hopes that a more familiar notation than Haskell or Scheme might make the principles more accessible. All the JavaScript code is explained, edit-

able, and executable on a web page [7].

Experiments with `Monad` and `MonadPlus` axioms served as a first test. Hinze's introduction to Haskell [8], Hutton's book [4], and examples implemented with `parsec` [5] suggested that parsing is a pedagogically useful application of monads. A homework assignment in a Haskell course [10] indicated that a monadic interpreter for a small programming language would be a realistic example. However, even in JavaScript the `andThen`/`orElse` chains quickly became incomprehensible, further illustrating the powerful synergy between Haskell's `do` notation and the `Monad` class. The monadic notation was designed for quick preprocessing, bootstrapped into JavaScript manually, and because of the `nodes` phrase it is now a good demonstration for grammars that this approach to parsing can handle.

The preprocessor and the interpreter work well in Rhino [11] but only Firefox among about a dozen tested browsers does not restrict the call stack depth for JavaScript. This limit in the other browsers makes preprocessing and interpretation fail, which in turn motivated moving to Java to avoid the artificial restrictions. As luck (or a very loosely coupled design) would have it, the monadic notation fit Java just as well as JavaScript, i.e., all it took to bootstrap into Java was to modify the JavaScript-based preprocessor to output Java rather than JavaScript. Generalizing, if one wants to port a parser (and the preprocessor) built using this technique to a new host language, it is likely that only the `Parser` and `ParserFactory` classes have to be translated and the preprocessor output might have to be tweaked a bit.

There is a performance penalty, but functional parsing handles grammars which the more traditional LR(1) and LL(1) systems cannot. As a result, the serendipitous question has finally resulted in an actual product. The monadic notation resembles `JavaCC`'s mix of grammar artifacts and Java code for semantic actions [12], but it was also possible to map the input languages used by the `oops3` parser generator [6] to the functional parser framework. As a result, even EBNF annotated with a bare minimum of regular expressions for tokens and class names for tree nodes can be compiled into a complete front end — scanner, parser, tree builder, and tree classes — available in a nearby web browser [13].

5. REFERENCES

(1) Haskell home page, http://haskell.org
(2) Haskell language definition, sections 8 and 20, http://haskell.org/onlinereport/
(4) G. Hutton, Programming in Haskell, Cambridge University Press, 2007
[6] A. Schreiner, *oops3* home page,

[7] A. Schreiner, Monadic Parsing using JavaScript,
http://www.cs.rit.edu/~ats/projects/jsm/paper.xml

[8] R. Hinze, Haskell Kurs,
http://www.informatik.uni-bonn.de/~ralf/teaching/Hskurs_7.html#SEC49

[9] D. Leijen, *parsec* home page,
http://legacy.cs.uu.nl/daan/parsec.html

[10] A. Schreiner, programming assignment,

[11] Rhino home page,
http://www.mozilla.org/rhino/

[12] JavaCC home page,
https://javacc.dev.java.net/

[13] A. Schreiner, Web Compiler Service home page,