Bashing Haskell: Reimplementing the Parsec Library Inside the Unix Shell

~~Functional~~ Imperative Pearl

Mike Izbicki
University of California Riverside
mike@izbicki.me

Abstract

We introduce the Parsed suite of Unix command line tools. Parsed improves the classic sed program by incorporating ideas from the popular `parsec` Haskell library. In particular, the Unix pipe operator (`|`) corresponds exactly to Haskell's Applicative bind `(*>)`. The resulting syntax is both intuitive and powerful. The original syntax used by sed can match only regular languages; but our improved syntax can match any context sensitive language.

Keywords unix, sed, parsing, parsec, Haskell, applicative, monad

1. Introduction

The stream editor (sed) is one of the oldest and most widely used Unix utilities. Unfortunately, it is a monolithic beast. It fails to live up to the Unix philosophy of "do one thing well." The problem is that sed tries to implement all regular expression features in a single executable. This ruins composibility. For example, let's say I have a simple sed command for deleting email addresses. Something like:

```
sed 's/[a-zA-Z0-9.]@[a-zA-Z0-9.]/xxx@xxx/g'
```

But in the file I'm working on, I only want to delete an email address if the line starts with the words `CONFIDENTIAL`. This should not be a hard task. We *should* be able to write another sed command for finding lines that begin with `CONFIDENTIAL`, then combine these two commands. But this is not possible using standard techniques. To solve our problem, we must rewrite an entirely new sed command from scratch. Within that command, we copy-and-paste our previously tested command:

```
sed 's/(CONFIDENTIAL.*)[a-zA-Z0-9.]@[a-zA-Z0-9.]/
    $1xxx@xxx/g'
```

Real world experience suggests that this practice is a leading source of bugs for the modern sed programmer.[1] In this paper, we simplify shell based parsing by introducing techniques from functional programming into the Unix shell. Our Parsed library combines the best of sed with Haskell's excellent Parsec library.

Parsec is a simple Haskell library that makes parsing easy and fun. It provides a small set of simple higher order functions called combinators. By combining these combinators, we can create programs capable of parsing complex grammars. The Parsed library recreates these combinators within the Unix terminal. In particular, each combinator corresponds to either a shell script or shell function. We then combine these combinators using the standard Unix pipe. There are three combinators of particular importance: `match`, `some`, and `choice`. With these combinators, we can match any regular expression. But unlike sed, our combinators can take advantage of the shell's built-in expressivity to match any context sensitive language!

[1] Can you spot the bug in the command?

In the remainder of this paper, we give a brief tutorial on how to construct your own parsers using Parsed. We recommend that you install Parsed and follow along with our examples. Installing is easy. Just clone the git repo:

```
$ git clone https://github.com/mikeizbicki/parsed
```

And add the resulting `parsed/scripts` folder to your `PATH`:

```
$ export PATH="$(pwd)/parsed/scripts:$PATH"
```

That's it! You're now ready to start parsing!

2. Matching strings

The most basic combinator is `match`. It's easiest to see how it works by example. Throughout this tutorial, we will first present the examples, and then their explanation.

```
$ match hello <<< "goodbye"
$ echo $?
1
```

`match` takes a single command line argument, which is the string our parser is trying to match. In the above example, this is the string `hello`. The text to be parsed comes from stdin. In the above example, we use bash's built-in `<<<` syntax, which redirects the contents of the following string (`goodbye`) to stdin. The exit code of the previously run program is stored in the shell variable `$?`. In the above example, our parser failed (the string `hello` does not match the string `goodbye`), so `$?` contains a non-zero value.

Now let's see what a successful parse looks like:

```
$ match hello <<< "hello"
hello
$ echo $?
0
```

When `match` succeeds, the matched text gets printed to stderr. This is where the output string `hello` comes from in the above example. Since parsing succeeded, `match` returned an exit code of 0. As you can see, `match` is a very simple parser. It is normally combined with other parsers.

3. Combining parsers

We combine parsers using the standard unix pipe (`|`). For example, let's create a parser that first matches the string `hello` and then matches the string `world`:

```
$ (match hello | match world) <<< "helloworld"
helloworld
$ echo $?
0
```

Script 1 `match`

```
#!/bin/sh

# check for the correct number of arguments
if [ -z "$1" ] || [ ! -z "$2" ]; then
    echo "match requires exactly one argument"
    exit 255
fi

# When reading from stdin, the shell ignores the
    contents of the IFS variable. By default,
    this is set to whitespace. In order to be
    able to read whitespace, we must set IFS to
    nothing.
IFS=''

# read in exactly as some characters as are in
    the first command line argument
read -rn "${#1}" in

# if we parse correctly
if [ "$1" = "$in" ]; then
    # print the parsed string to stderr
    echo -n "$in" 1>&2

    # forward the unparsed contents of stdin
    cat

    # signal that parsing succeeded
    exit 0

else
    # parsing failed
    exit 1
fi
```

Parsing succeeds for both calls to `match`, so parsing succeeds overall. To see how piping combines parsers, we need to take a more careful look at the `match` script. Like all combinators in the Parsed library, `match` is written in POSIX compliant shell. The full source code is shown in Script 1 above.

We've already seen that when `match` succeeds, the matched string is printed to stderr; but additionally, any unparsed text is printed to stdout. This is demonstrated by the following two tests:

```
$ match hello <<< "helloworld" 1> /dev/null
hello
$ match hello <<< "helloworld" 2> /dev/null
world
```

As a reminder, by putting a number in front of the output redirection operator >, we redirect the contents of that file descriptor to the specified file. File descriptor 1 corresponds to stdout and 2 to stderr. So the first command above discards stdout; it shows only the text that was successfully parsed. The second command above discards stderr; it shows only the text that is sent to any subsequent parsers.

It is possible for the first parser to succeed and the second to fail. In this case, the succeeding parsers still print their output to stderr, but parsing fails overall:

```
$ (match hello | match world) <<< "hellogoodbye"
hello
$ echo $?
1
```

Concatenating two `match` parsers is relatively uninteresting. We could have just concatenated the arguments to `match`! So now let's introduce a new combinator called `eof` which matches the end of file. That is, `eof` will succeed only if the stdin buffer has been closed because there is no more input to parse. The source code for `eof` is much simpler than for `match`, and is shown in Script 2.

Script 2 `eof`

```
#!/bin/sh

# Try to read a single character into the
    variable $next. If next is empty, we're at
    the end of file, so parsing succeeds.
IFS=
read -n 1 next
if [ -z $next ]; then exit 0; else exit 1; fi
```

These next two examples demonstrate the effect of the `eof` combinator. First, we try matching the string `hello` on the input `hellohello`:

```
$ match hello <<< "hellohello"
hellohello
$ echo $?
0
```

Parsing succeeds; we print the contents of the parsed text (`hello`) to stderr; and we print the remaining content to be parsed (`hello`) to stdout. If, however, we apply the `eof` combinator:

```
$ (match hello | eof) <<< "hellohello"
hello
$ echo $?
1
```

Parsing now fails because the input was too long. Shortening the input again causes parsing to succeed:

```
$ (match hello | eof) <<< "hello"
hello
$ echo $?
0
```

Now that we know how to combine two simple parsers, we're ready for some more complex parsers.

4. Iterating `some` parsers

The `some` combinator lets us apply a parser zero or more times. `some` corresponds to the Kleene star (`*`) operator used in `sed` and most other regular expression tools. `some` takes a single command line argument, which is the parser we will be applying zero or more times. For example:

```
$ (some "match hello" | eof) <<< "hellohello"
hellohello
$ echo $?
0
```

The above example succeeds because the `some "match hello"` parser consumes *both* occurrences of `hello`. As already mentioned, `some` need not consume any input:

```
$ some "match hello" <<< ""
$ echo $?
0
$ some "match hello" <<< "goodbye"
$ echo $?
0
```

In fact, the `some` used by itself can never fail—it will always just parse the empty string. In order to force failures, we must concatenate `some` with another parser like so:

```
$ (some "match hello" | eof) <<< "goodbye"
$ echo $?
1
```

Script 3 `some`

```
#!/bin/sh

# check for the correct number of arguments
if [ -z "$1" ] || [ ! -z "$2" ]; then
    echo "many␣requires␣exactly␣one␣argument"
    exit 255
fi

# put the contents of stdin into a variable so we
     can check if it's empty
stdin=$(cat)

# if we still have input and parsing succeeded
if [ ! -z "$stdin" ] && stdout=`eval "$1" <<< "
    $stdin"`; then

    # run this parser again
    "$0" "$1" <<< "$stdout"
else

    # stop running this parser
    cat <<< "$stdin"
fi
```

Unfortunately, the `some` combinator breaks the ability to use POSIX pipes for concatenation as we did above. Consider this example:

```
$ (match hello | some "match␣hello") <<< ""
$ echo $?
0
```

Our first `match` parser failed because there was no input. Then we call the `some` parser. There is still no input, so `some` succeeds. Since `some` was the last command to execute, `$?` contains its exit code which is 0.

The problem is that the standard POSIX `$?` variable has the wrong behavior. We need a variable that will report if any of the commands in the pipe chain failed. There is no POSIX compliant way to do this. But more modern shells like bash offer a simple fix. The command

```
$ set -o pipefail
```

modifies the semantics of the `$?` variable so that it contains zero only if all commands in the pipe chain succeed; otherwise, it contains the exit code of the first process to exit with non-zero status. This command need only be run once per `bash` session. Thereafter, we can rerun the incorrect example above to get the correct output:

```
$ (match hello | some "match␣hello") <<< ""
$ echo $?
1
```

The code for the `some` combinator is shown in Script 3 above.

5. Custom combinatorial explosion

Most regular expression libraries offer a primitive combinator + that matches one or more occurrences of a previous parser. Parsed does not have such a primitive operator because it is easy to build it using only the `|` and `some` combinators. We call the resulting operator `many`, and we implement it by creating a bash function:

```
$ many() { $1 | some "$1"; }
```

Recall that the `$1` variable will contain the first parameter to the `many` function. In this case, that parameter must be a parser. We can use our new combinator to simplify the examples from the previous section:

Script 4 `many`

```
#!/bin/bash
set -o pipefail
$1 | some "$1"
exit $?
```

```
$ many "match␣hello" <<< ""
$ echo $?
1
$ many "match␣hello" <<< "hello"
hello
$ echo $?
0
```

Unfortunately, Bash functions do not last between sessions. If we want something more permanent, we can create a script file instead. In fact, the `many` combinator is so useful that it comes built-in to the Parsed library. You can find its source code in Script 4 above. When the script file starts, it will not inherit the settings of its parent process in the same way that our `many` function did. Therefore, we must specifically re-setup our non-POSIX pipes at the beginning of every script that uses them.

6. The program of choice

The last combintor we need for parsing regular languages is called `choice`. The `match` and `many` combinators required exactly one input, but choice takes an arbitrary number of input parameters. Each parameter is a parser. For each of these parsers, `choice` applies it to stdin. If it succeeds, then `choice` returns success. If it fails, then `choice` goes on to the next parameter. If all parsers fail, then `choice` fails as well.

Here's a simple example using just the match combinator:

```
$ choice "match␣hello" "match␣hola" <<< hello
hello
$ echo $?
0
$ choice "match␣hello" "match␣hola" <<< hola
hola
$ echo $?
0
$ choice "match␣hello" "match␣hola" <<< goodbye
$ echo $?
1
```

Our parser succeeds when the input was either `hello` or `hola`, but fails on any other input.

7. Free your context

In the intro we claimed that Parsed is strictly more powerful than sed. We now demonstrate this feature by parsing a context free language.[2] Sed only supports regular expressions. The reason Parsed has this extra power is because we can define our own recursive parsers using standard shell syntax.

To demonstrate this capability, we will write a parser that checks for balanaced parenthesis. First, let's define a small combinator `paren` that takes a single parser as an argument and creates a parser that succeeds only when the parameter is surrounded by parentheses:

```
$ paren() { match "(" | $1 | match ")"; }
```

[2] Context sensitive languages are left as an exercise to the reader.

Script 5 `choice`

```
#!/bin/bash

# We need to store the contents of stdin
    explicitly to a file.  When we call one of
    our candidate parsers, it will consume some
    of the input form stdin. We need to restore
    that input before calling the next parser.
stdin=$(tempfile)
cat >"$stdin"

# If stdin is empty, then we're done with
    recursion; parsing succeeded
if [ ! -s "$stdin" ]; then
    exit 0
fi

# For similar reasons, we'll need to store the
    output of our parsers.
stdout=$(tempfile)
stderr=$(tempfile)

# For each parser passed in as a command line arg
for cmd in "$@"; do

    # Run the parser; if it succeeds, then pass
    on its results
    if $(eval "$cmd" <"$stdin" 1>"$stdout" 2>"
    $stderr") ; then
        cat "$stderr" >&2
        cat "$stdout"
        exit 0
    fi
done

# All parsers failed, so we failed
exit 1
```

And let's test it:

```
$ paren "match␣hello" <<< "hello"
$ echo $?
1
$ paren "match␣hello" <<< "(hello)"
(hello)
$ echo $?
0
```

We will use `paren` to write a combinator `maybeparen` that accepts whether or not the parameter parser is surrounded by parenthesis. Here is a reasonable first attempt:

```
$ maybeparen() { choice "$1" "paren␣$1"; }
```

Unfortunately, this doesn't work due to scoping issues with bash. When we run our command, we get an error:

```
$ maybeparen "match␣a" <<< "(a)"
choice: line 7: paren: command not found
```

We get this error because the `choice` combinator is its own shell script. When this script gets executed, a new shell process starts with a clean set of environment variables. In the context of this subprocess, the `paren` function we created above doesn't exist.

There are two ways to solve this problem. The simplest is to use the following bash-specific syntax:

```
$ export -f paren
```

This command tells the running `bash` shell that any subshells it spawns should also have access to the `paren` function. Now when we try running our previous tests:

Script 6 `parens`

```
#!/bin/bash
choice "$1" "match␣'('␣|parens␣\"$1\"␣|match␣')'"
```

```
$ maybeparen "match␣a" <<< "a"
a
$ echo $?
0
$ maybeparen "match␣a" <<< "(a)"
$ echo $?
1
```

We still get a parse error! What's happening is that we actually defined the `maybeparen` function incorrectly. Here is the correct definition:

```
$ maybeparen() { choice "$1" "paren␣\"$1\""; }
```

The only difference is that the correct definition surrounds our `$1` parameter with quotation marks. This ensures that the entire value of the variable gets passed as a single parameter to the `paren` combinator. Because these quotation marks are within quotation marks, they must be escaped with backslashes.

We are now ready to define a combinator `parens` that matches any number of balanced parentheses. In order to avoid the need for a set of triply nested quotation marks, we will put the `parens` combinator within a script file (instead of a function) and we will manually inline our call to the `paren` combinator. Script 6 above contains the final result. And now let's test our creation:

```
$ parens "match␣a" <<< "(((a)))"
(((a)))
$ echo $?
0
$ parens "match␣a" <<< "(((b)))"
$ echo $?
1
$ parens "match␣a" <<< "(((a"
$ echo $?
1
```

We've successfully parsed a context free language :)

8. Discussion

This tutorial has provided only a very brief introduction to the capabilities of our Parsed library. Additional under-documented features include: (1) We can use shell variables to simulate Haskell's `Monad` type class. In particular, the shell code:

```
var=$(func)
```

is equivalent to the haskell do-notation code:

```
var <- func
```

In fact, all shell commands can be thought of as being within the `IO` monad wrapped within the `ParsecT` transformer. This leads us to our next under-documented feature. (2) Arbitrary commands can be used in parsers. Want to punish your users when they make syntax errors? Just add an `rm -rf *` at the appropriate place in your combinators. In Parsed, we are no longer confined to the limits of the Haskell type system—we can exploit bash's flexibility to literally do whatever we want! The Unix shell is just *wonderful* like that.

References

[1] Daan Leijen and Erik Meijer. Parsec: Direct style monadic parser combinators for the real world. 2002.