Inside 736-131

Existential Pontification and Generalized Abstract Digressions

Parsec: “try a <|> b” considered harmful

tl;dr The scope of backtracking try should be minimized, usually by placing it inside the definition of a parser.

Have you ever written a Parsec parser and gotten a really uninformative error message?

```
"test.txt" (line 15, column 7):
unexpected 'A'
expecting end of input
```

The line and the column are randomly somewhere in your document, and you're pretty sure you should be in the middle of some stack of parser combinators. But wait! Parsec has somehow concluded that the document should be ending immediately. You noodle around and furthermore discover that the true error is some ways *after* the actually reported line and column.

You think, “No wonder Parsec gets such a bad rep about its error handling.”

Assuming that your grammar in question is not too weird, there is usually a simple explanation for an error message like this: the programmer sprinkled their code with too many backtracking `try` statements, and the backtracking has destroyed useful error state. In effect, at some point the parser failed for the reason we wanted to report to the user, but an enclosing `try` statement forced the parser to backtrack and try another (futile possibility).

This can be illustrated by way of an example. A Haskeller is playing around with parse combinators and decides to test out their parsing skills by writing a parser for Haskell module imports:

```
stmt ::= import qualified A as B
       | import A
```

Piggy-backing off of Parsec’s built in token combinators (and the sample code), their first version might look something like this:

```
import Text.Parsec
import qualified Text.Parsec.Token as P
import Text.Parsec.Language (haskellDef)

data Stmt = QualifiedImport String String | Import String
    deriving (Show)

pStmt = pQualifiedImport <|> pImport

pQualifiedImport = do
    reserved "import"
    reserved "qualified"
    i <- identifier
    reserved "as"
    i' <- identifier
    return (QualifiedImport i i')

pImport = do
    reserved "import"
    i <- identifier
    return (Import i)

lexer = P.makeTokenParser (haskellDef
    { P.reservedNames = P.reservedNames haskellDef ++ ["qualified", "as"] })
identifier = P.identifier lexer
reserved = P.reserved lexer

parseStmt input = parse (pStmt >> eof) "(unknown)" input
```

Unfortunately, the parser doesn't work for regular imports—they get this error message:

```
*Main> parseStmt "import Foo"
Left "(unknown)" (line 1, column 8):
unexpected "F"
expecting "qualified"
```

After a little Googling, they discover that Parsec doesn't backtrack by default. Well, that's fine; why not just insert a try into the parser.

```
pStmt = try pQualifiedImport <|> pImport
```

This fixes both parses and suggests the following rule for writing future parsers:

> If I need choice over multiple parsers, but some of these parsers might consume input, I better tack a `try` onto each of the parsers, so that I can backtrack.

Unbeknownst to the user, they have introduced bad error reporting behavior:

```
*Main> parseStmt "import qualified Foo s B"
Left "(unknown)" (line 1, column 17):
unexpected reserved word "qualified"
expecting letter or digit or "#"
```

Wait a second! The error we wanted was that there was an unexpected

identifier `s`, when we were expecting `as`. But instead of reporting an error when this occurred, Parsec instead *backtracked*, and attempted to match the `pImport` rule, only failing once that rule failed. By then, the knowledge that one of our choice branches failed had been forever lost.

How can we fix it? The problem is that our code backtracks when we, the developer, know it will be futile. In particular, once we have parsed `import qualified`, we know that the statement is, in fact, a qualified import, and we shouldn't backtrack anymore. How can we get Parsec to understand this? Simple: *reduce the scope of the try backtracking operator:*

```
pStmt = pQualifiedImport <|> pImport

pQualifiedImport = do
    try $ do
        reserved "import"
        reserved "qualified"
    i <- identifier
    reserved "as"
    i' <- identifier
    return (QualifiedImport i i')
```

Here, we have moved the `try` from `pStmt` into `pQualifiedImport`, and we only backtrack if `import qualified` fails to parse. Once it parses, we consume those tokens and we are now committed to the choice of a qualified import. The error messages get correspondingly better:

```
*Main> parseStmt "import qualified Foo s F"
Left "(unknown)" (line 1, column 22):
unexpected "s"
expecting "as"
```

The moral of the story: The scope of backtracking try should be minimized, usually by placing it inside the definition of a parser. Some amount of cleverness is required: you have to be able to identify how much lookahead is necessary to commit to a branch, which generally depends on *how* the parser is used. Fortunately, many languages are constructed specifically so that the necessary lookahead is not too large, and for the types of projects I might use Parsec for, I'd be happy to sacrifice this modularity.

Another way of looking at this fiasco is that Parsec is at fault: it shouldn't offer an API that makes it so easy to mess up error messages—why can't it automatically figure out what the necessary lookahead is? While a traditional parser generator can achieve this (and improve efficiency by avoiding backtracking altogether in our earlier example), there are some fundamental reasons why Parsec (and monadic parser combinator libraries like it) cannot automatically determine what the lookahead needs to be. This is one of the reasons (among many) why many Haskellers prefer faster parsers which simply don't try to do any error handling at all.

Why, then, did I write this post in the first place? There is still a substantial amount of documentation recommending the use of Parsec, and a beginning Haskeller is more likely than not going to implement their first parser in Parsec. And if someone is going to write a Parsec parser, you might as well spend a little time to limit your backtracking: it can make working with Parsec parsers a *lot* more pleasant.

- May 17, 2014
- Haskell

11 Responses to “Parsec: “try a <|> b” considered harmful”

1.  *Greg Weber* says:
   May 17, 2014 at 11:46 pm

   What Applicative parsing libraries are available and what would you recommend?

2.  *Edward Z. Yang* says:
   May 18, 2014 at 2:15 am

   Hm! So there definitely has been a ground-shift with regards to the idiomatic way to write parsers (applicatively instead of using monads), which is what I was alluding to, but when I saw your question and checked the most popular packages, it wasn’t obviously the case that they were doing something more efficient when you only used the applicative interface. Certainly attoparsec is not doing anything better, and from a glance, trifecta, parsimony and polyparse don’t do anything special either. So I think I misspoke, and I will reword the sentence accordingly.

   A more academic parsing library which is more efficient when you use the applicative instance is uu-parsinglib (http://hackage.haskell.org/package/uu-parsinglib) although I do not know why it has not caught on in the wider zeitgeist. (Breadth-first parsing can be a bit idiosyncratic.)

3.  *Ben Edwards* says:
   May 18, 2014 at 9:43 am

   The big efficiency win with trifecta is the slicing conbinators. When you know you are thowing the intermediate productions, replace with unit! Of course, this has nothing to do with being monad or applicative. I just thought I’d toss it out there.

4.  *Sami Liedes* says:
[May 18, 2014 at 9:48 am]

A while ago I needed to parse something substantial and, being new to Haskell parser combinators, I started with parsec. Eventually I tried attoparsec, and contrary to what everybody says, to me its error messages were generally better and it was easier to use than parsec. I no longer needed to sprinkle my code with magic ‘try’ in just the right places, and IIRC attoparsec always identified in error messages correctly the set of tokens it would have accepted and the token it got instead, which is just what parsec loses when you use ‘try’. The only (admittedly substantial) drawback was that the error messages could not show the position of the error in the input.

5.  *Greg Weber* says:
[May 18, 2014 at 11:05 am]

uu-parsing hasn’t caught on because it does not scale down to new users on small tasks. In the time it would take me to wrap my head around how to use this lib, I will already be done with the job with Parsec (even if I had to just learn Parsec) because the Parsec documentation is good and the API is easy to understand. I hope uu-parsing will get an API makeover and some good documentation.

6.  *Edward Kmett* says:
[May 18, 2014 at 11:57 am]

Note: Trifecta explicitly will not move your error messages when backtracking for try. This means you have to put another expected token on.

The usual idiom is to treat the thing you are forced to try as a lexeme and blame the whole structure.

pStmt = try pQualifiedImport <?> “qualified import”
pImport

The benefit there is if the other parser fails you can get both in the set of expected tokens at the point to which it had to backtrack.

7.  *Edward Kmett* says:
[May 18, 2014 at 11:58 am]

There was a <?> in there. =)

8.  *Anonymous* says:
[May 18, 2014 at 1:36 pm]

You can try parsek instead. For errors, it has an option to return the error for the longest correctly parsed alternative (not just the rightmost(?) like parsec).

9.  *Edward Z. Yang* says:
[May 19, 2014 at 6:57 am]

Ben: So /that's/ what a slicing combinator is! It is not at all clear from the Haddock documentation, "Run a parser, grabbing all of the text between its start and end points" that this is the intended use-case. If you manually cut, much in the same way I'm doing here with 'try', then yes, applicative structure is not needed.

Anonymous: Which is… apparently in the 'Encode' package? The source paper suggests that some sort of breadth-first strategy is also being employed here.

There was also discussion on Reddit: http://www.reddit.com/r/haskell/comments/25ulsv/ezyang_try_a_b_considered_harmful/

10.  *massysett* says:
[May 20, 2014 at 1:15 pm]

Your last paragraph seems to suggest (if I may paraphrase) "well, gee, Parsec is not that great, but a lot of docs say use it, so here's how to use it, but really you might want to look at something faster that doesn't even bother with errors because those faster libraries backtrack automatically."

Parsec is not just a default choice that has a lot of documentation. Rather, for many jobs it is the best choice. In many contexts attoparsec is utterly useless precisely because it does not deliver decent error messages.

I have a parser which needs to deliver good error messages, because it parses handwritten files. If there's an error the user needs to know where it is so he can fix it. But sure, it would be nice if the parser were faster. So I tried attoparsec. I figured I could use attoparsec if it completes successfully but, because its error messages are useless, I could fall back to Parsec and parse the file again and get an error message for those cases in which the parse fails.

It turned out that for this particular grammar attoparsec was not much faster! There may have been something like a 15% speed boost but nothing that made it worthwhile to maintain two parsers. I also tried happy and alex and saw no appreciable speed gain there either, and their error messages are nowhere near as good as Parsec's.

It seems to me that Parsec has become a whipping boy for complaints of (1) "try is too hard" and (2) "it's too slow" when (1) you need to control backtracking to get good error messages and (2) for many grammars it's not appreciably slower!

11.  *Krakrjak* says:
[June 2, 2014 at 1:46 pm]

Massysett, 0.12.0 of attoparsec was just released over the weekend. The change log reports some impressive speed ups. Might be a good time to revisit your dual parser version of the code.

Leave a Comment

Name (Optional):

Comment:

Post Comment

« Previous Post Next Post »