An introduction to parsing text in Haskell with Parsec

Parsec makes parsing text very easy in Haskell. I write this as much for myself as for anyone else to have a tutorial and reference which starts from the ground up and works through how each function can be used with examples all the way.

First off, why would you use Parsec as opposed to things like regular expressions for parsing content? Coming from other languages, splitting content into arrays and processing ever smaller chunks using regular expressions and the like can be quite common practise. In Haskell we can go down that route too, but now I've seen the light with Parsec I want to introduce you to a better approach.

Most tutorials get stuck in with a complete example straight off the bat, but I'm going to present the different functions one by one so that they can be used as a reference of reminder (to me as much as anyone!) of how things work. I'll try to keep examples relatively self contained, so it shouldn't be hard to skip to bits, but keep in mind the basic setup I do first. I have also put all of the example code into a file right here that is ready to be `:load`ed straight into `ghci` and played with.

The Basics

Parsec works its way along some stream of text from beginning to end, attempting to match the stream of inputs to some rule or a set of rules. Parsec is also monadic, so we can piece together our different rules in sequence using the convenient do notation. As a general overview, rules work by consuming a character at a time from the input and determining whether they match or not, so when you piece a number of rules in sequence, each rule will consume part of the input until you have no input left, run out of rules, or one of your rules fails to match (resulting in an error).

Let's start with a very basic setup. I import `Parsec` qualified so it's plainly obvious when I'm using `Parsec` functions, and also import `Control.Applicative` so we can play with parsing things in the applicative style later on, and make a short alias for `parseTest` for use in the examples:

```
-- I import qualified so that it's clear which
-- functions are from the parsec library:
import qualified Text.Parsec as Parsec

-- I am the error message infix operator, used later:
import Text.Parsec ((<?>))

-- Imported so we can play with applicative things later.
-- not qualified as mostly infix operators we'll be using.
import Control.Applicative

-- Get the Identity monad from here:
import Control.Monad.Identity (Identity)

-- alias Parsec.parse for more concise usage in my examples:
parse rule text = Parsec.parse rule "(source)" text
```

About Me

My name is James Wilson. I studied Cognitive Science at Hertfordshire University, graduating with a first class degree, and went on to complete a Ph.D in Developmental Robotics at Aberystwyth University. I've been programming since I was a kid, and am always trying out new languages. I have a particular interest in biologically inspired machine learning algorithms, and I love making things. Don't hesitate to get in touch if you want to know more.