Если вы видите что-то необычное, просто сообщите мне.

Parsing Primer: Gherkin Syntax

Haskell is a truly awesome language for parsing. Haskell expressions tend to compose in simple ways with very clearly controlled side effects. This provides an ideal environment in which to break down parsing into simpler tasks. Thus there are many excellent parsing libraries out there.

In this series, we'll be taking a tour of some of these libraries. But before we look at specific code, it will be useful to establish a common example for what we're going to be parsing. In this first part, I'll introduce Gherkin Syntax, the language behind the Cucumber framework. We'll go through the language specifics, then show the basics of how we set ourselves up for success in Haskell.

If you're already familiar with Gherkin syntax, feel free to move on to part 2 of this series! Or if you want to challenge yourself with more libraries to learn, you can download our Production Checklist. It'll give you a survey of libraries for many different tasks!

Like some of our other series, there's a Github Repository that has all the code for these tutorials! You can observe some Gherkin examples in this directory. We'll compose our Haskell types in this module.

GHERKIN BACKGROUND

Cucumber is a framework for Behavior Driven Development. Under BDD, we first describe all the general behaviors we want our code to perform in plain language. This paradigm is an alternative to Test Driven Development. There, we use test cases to determine our next programming objectives. But BDD can do both of these if we can take behavior descriptions and automatically create tests from them! This would allow less technical members of a project team to effectively write tests!

The main challenge of this is formalizing a language for describing these behaviors. If we have a formal language, then we can parse it. If we can parse it into a reasonable structure, then we can turn that structure into runnable test code. This series will focus on the second part of this problem: turning Gherkin Syntax into a data structure (a Haskell data structure, in our case).

GHERKIN SYNTAX

Gherkin syntax has many complexities, but for these articles we'll be focusing on the core elements of it. The behaviors you want to test are broken down into a series of features. We describe each feature in its own .feature file. So our overarching task is to read input from a single file and turn it into a Feature object.

We begin our description of a feature with the Feature keyword (obviously). We'll give it a title, and then give it an indented description (our example will be a simple banking app):

```
Feature: Registering a User
  As a potential user
  I want to be able to create an account with a username,
    email and password
  So that I can start depositing money into my account
```

Each feature then has a series of scenarios. These describe specific cases of what can happen as part of this feature. Each scenario begins with the Scenario keyword and a title.

```
Scenario: Successful registration
  ...

Scenario: Email is already taken
  ...

Scenario: Username is already taken
  ...
```

Each scenario then has a series of Gherkin statements. These statements begin with one of the keywords Given, When, Then, or And. You should use Given statements to describe pre-conditions of the scenario. Then you'll use When to describe the particular action a user is taking to initiate

the scenario. And finally, you'll use Then to describe the after effects.

```
Scenario: Email is already taken
  Given there is already an account with the email "test@test.com"
  When I register an account with username "test",
    email "test@test.com" and password "1234abcd!?"
  Then it should fail with an error:
    "An account with that email already exists"
You can supplement any of these cases with a statement beginning with And.

Scenario: Email is already taken
  Given there is already an account with the email "test@test.com"
  And there is already an account with the username "test"
  When I register an account with username "test",
    email "test@test.com" and password "1234abcd!?"
  Then it should fail with an error:
    "An account with that email already exists"
  And there should still only be one account with
    the email "test@test.com"
```

Gherkin syntax does not enforce that you use the different keywords in a semantically sound way. We could start every statement with Given and it would still work. But obviously you should do whatever you can to make your tests sound correct.

We can also fill in statements with variables in angle brackets. We'll then follow the scenario with a table of examples for those variables:

```
Scenario: Successful Registration
  Given There is no account with username <username>
    or email <email>
  When I register the account with username <username>,
    email <email> and password <password>
  Then it should successfully create the account
    with <username>, <email>, and <password>
  Examples:
    | username | email              | password      |
    | john doe | john@doe.com       | ABCD1234!?    |
    | jane doe | jane.doe@gmail.com | abcdefgh1.aba |
    | jackson  | jackson@yahoo.com  | cadsw4ll0p/   |
```

We can also create a Background for the whole feature. This is a scenario-like description of preconditions that exist for every scenario in that feature. This can also have an example table:

```
Feature: User Log In
  ...

Background:
  Given: There is an existing user with username <username>,
    email <email> and password <password>
  Examples:
    | username | email              | password      |
    | john doe | john@doe.com       | ABCD1234!?    |
    | jane doe | jane.doe@gmail.com | abcdefgh1.aba |
```

And that's the whole language we're going to be working with!

HASKELL DATA STRUCTURES

Let's appreciate now how easy it is to create data structures in Haskell to represent this syntax. We'll start with a description of a Feature. It has a title, description (which we'll treat as a list of multiple lines), the background, and then a list of scenarios. We'll also treat the background like a "nameless" scenario that may or may not exist:

```
data Feature = Feature
  { featureTitle :: String
  , featureDescription :: [String]
  , featureBackground :: Maybe Scenario
  , featureScenarios :: [Scenario]
  }
```

Now let's describe what a Scenario is. It's main components are its title and a list of statements. We'll also observe that we should have some kind of structure for the list of examples we'll provide:

```
data Scenario = Scenario
  { scenarioTitle :: String
  , scenarioStatements :: [Statement]
  , scenarioExamples :: ExampleTable
```

```
  }
```

This ExampleTable will store a list of possible keys as well as list of tuple maps. Each tuple will contain keys and values. At the scale we're likely to be working at, it's not worth it to use a full Map:

```
data ExampleTable = ExampleTable
  { exampleTableKeys :: [String]
  , exampleTableExamples :: [[(String, Value)]]
  }
```

Now we'll have to define what we mean by a Value. We'll keep it simple and only use literal bools, strings, numbers, and a null value:

```
data Value =
  ValueNumber Scientific |
  ValueString String |
  ValueBool Bool |
  ValueNull
```

And finally we'll describe a statement. This will have the string itself, as well as a list of variable keywords to interpolate:

```
data Statement = Statement
  { statementText :: String
  , statementExampleVariables :: [String]
  }
```

And that's all there is too it! We can put all these types in a single file and feel pretty good about that. In Java or C++, we would want to make a separate file (or two!) for each type and there would be a lot more boilerplate involved.

GENERAL PARSING APPROACH

Another reason we'll see that Haskell is good for parsing is the ease of breaking problems down into smaller pieces. We'll have one function for parsing an example table, a different function for parsing a statement, and so on. Then gluing these together will actually be slick and simple!

CONCLUSION

Now you can move on to part 2 where we'll actually look at how to start parsing this. The first library we'll use is the regex-applicative parsing library. We'll see how we can get a lot of what we want without even using a monadic context!

For some more ideas on parsing libraries you can use, check out our free Production Checklist. It will tell you about different libraries for parsing as well as a great many other tasks, from data structures to web APIs!

Revision #1
Created 2022-03-11 16:09:19 UTC by gasick
Updated 2022-03-11 17:11:16 UTC by gasick