A TASTE OF TAXONOMIES

Today we're going to talk about:

* The data structure known as a "taxonomy"

* How taxonomies are represented in CSV files

* How taxonomies are stored in relational databases

* How to validate taxonomies before we store them

The Problem

Let's start with a problem: As a software developer, you are presented with a CSV file that looks like this:

```
$> cat automobiles_taxonomy.csv

path,audience_size,cpm,buyable
"|General Motors","","",0
"|General Motors|Chevrolet",1000000,0.80,1
"|General Motors|Chevrolet|Spark",100000,0.15,1
"|General Motors|Chevrolet|Sonic",150000,0.20,1
"|General Motors|Chevrolet|Cruze",250000,0.20,1
"|General Motors|Chevrolet|Malibu",200000,0.25,1
"|General Motors|Chevrolet|Impala",175000,0.30,1
"|General Motors|Cadillac",150000,1.00,1
"|General Motors|Cadillac|Coupe",150000,0.30,1
"|General Motors|Cadillac|Sedan",150000,0.35,1
"|General Motors|Cadillac|Crossover",150000,0.40,1
"|General Motors|Cadillac|Escalade",150000,0.50,1
"|Toyota","","",0
"|Toyota|Corolla",800000,0.60,1
"|Toyota|Camry",600000,0.70,1
"|Toyota|Prius",400000,0.80,1
"|Toyota|RAV4",300000,0.50,1
```

You are told that the first column in this CSV file -- `path` -- holds hierarchical data about automobile manufacturers, their major car lines or "marques" and individual models. This first column has its own pipe-delimited format:

```
"|manufacturer|marque|model"
```

You are instructed to process this CSV data and insert it into a relational database table named `autos`. In the `autos` table, however, each row is uniquely identified by an `id` column and the hierarchical nature of the data has to be described by a combination of three columns -- `id`, `parent_id` and `name` -- like this:

```
 id | parent_id |      name      | audience_size | cpm  | is_actionable
----+-----------+----------------+---------------+------+---------------
  1 |           | General Motors |               |      | f
  2 |           | Toyota         |               |      | f
  3 |         1 | Chevrolet      |       1000000 | 0.80 | t
  4 |         1 | Cadillac       |        150000 | 1.00 | t
  5 |         2 | Corolla        |        800000 | 0.60 | t
  6 |         2 | Camry          |        600000 | 0.70 | t
  7 |         2 | Prius          |        400000 | 0.80 | t
  8 |         2 | RAV4           |        300000 | 0.50 | t
```

```
  9 |          3 | Spark              |         100000 | 0.15 | t
 10 |          3 | Sonic              |         150000 | 0.20 | t
 11 |          3 | Cruze              |         250000 | 0.20 | t
 12 |          3 | Malibu             |         200000 | 0.25 | t
 13 |          3 | Impala             |         175000 | 0.30 | t
 14 |          4 | Coupe              |         150000 | 0.30 | t
 15 |          4 | Sedan              |         150000 | 0.35 | t
 16 |          4 | Crossover          |         150000 | 0.40 | t
 17 |          4 | Escalade           |         150000 | 0.50 | t
```

The `name` column is the last part of the pipe-delimited string you see in the `path` column of the incoming CSV file. The `parent_id` column is the `id` number of the row in the database holding a `name` corresponding to the next-to-last part of the pipe-delimited string you see in the `path` column. Let's look at one row in the CSV file:

```
"|General Motors|Chevrolet|Spark",100000,0.15,1
```

Once this record is assigned `id 9` in the `autos` table and `name Spark`, you ask, "What is the `id` for `Chevrolet` in the `autos` table?", and you assign that value -- `3` to the `parent_id` column of the row whose `id` is `6`.

```
  9 |          3 | Spark              |         100000 | 0.15 | t
```

`Chevrolet`'s `parent_id`, in turn, becomes `1` because the `id` assigned to `General Motors` -- Chevrolet's "parent" -- is `1`.

You face a problem: Is the data in the CSV file truly hierarchical and, if it is and you get it into the database table, can you confirm that you preserved its hierarchical nature in the new format? Can you actually get from here to there?

To answer these questions, we first have to learn something about the data structure known as a **taxonomy**.

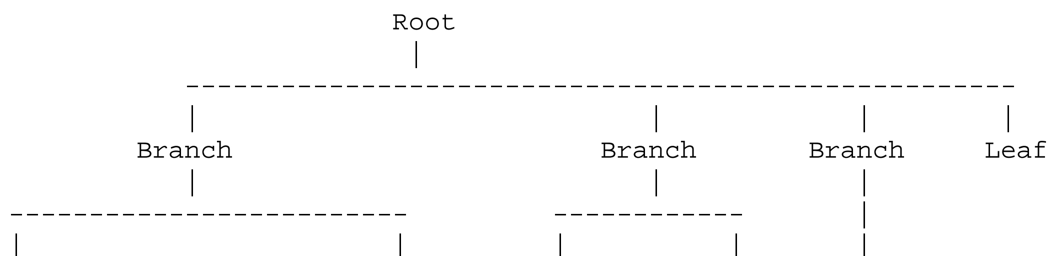
What Is a Taxonomy?

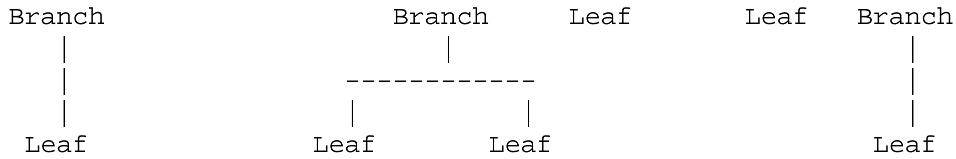
In today's presentation I hope to:

- Give you an introduction to taxonomies;
- Show you how taxonomies can be represented in plain-text CSV files; and
- Present some interesting use cases, including transforming one kind of taxonomy into another.

In this presentation I will define a **taxonomy** as a tree-like data structure (such as that shown in Diagram 1) with a root node, zero or more branch (child) nodes, and one or more leaf nodes.

Diagram 1:

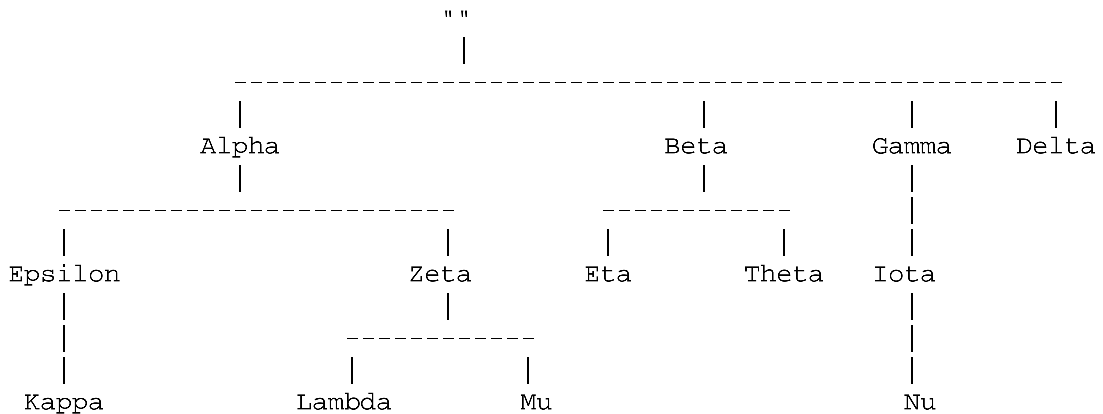




The root node and each branch node must have at least one child node, but leaf nodes have no child nodes. The number of branches between a leaf node and the root node is variable.

Let's make this structure a bit more human-friendly by giving names to the various nodes.

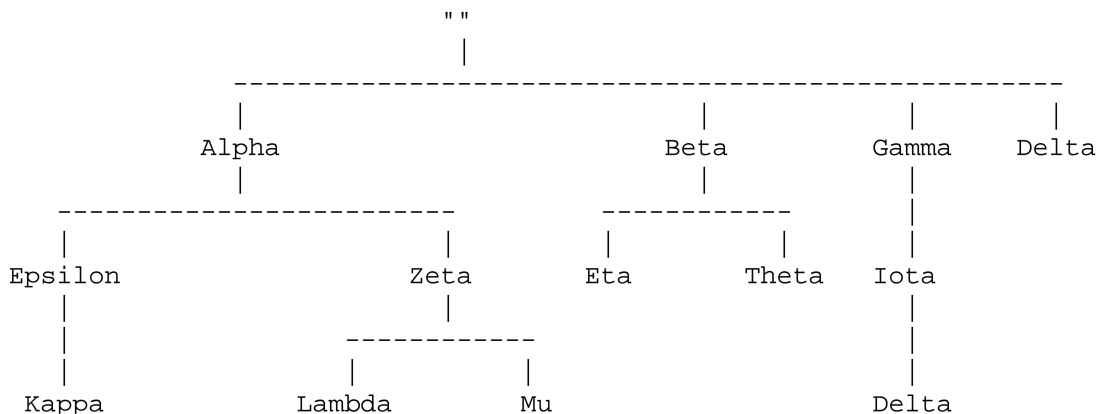
Diagram 2:



We can leave the root node unnamed for now. (You can think of that as "Mother Nature", "the Divinity" or "the Unnameable.")

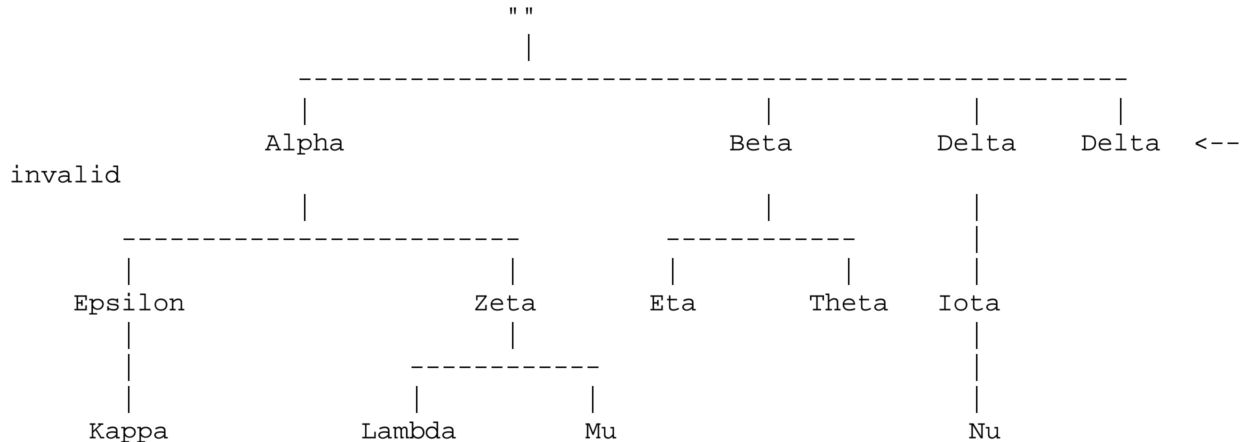
Looking at Diagram 2, you might well ask, "Do the nodes in the taxonomy have to have unique names?" The answer is: No, they don't -- with one exception. No two nodes which are children of the same parent can have the same name -- but it is okay for nodes which are children of different parents to be named the same. So Diagram 3 shows a legitimate taxonomy because the two nodes named "Delta" have different parents.

Diagram 3: (valid)



But Diagram 4 is **not** a legitimate taxonomy because two children of the root node both are named "Delta."

Diagram 4: (invalid)



A taxonomy may or may not represent some intrisic hierarchical structure within a data set. What's more important is that a taxonomy be **useful** in organizing human thinking about a data set. The most famous taxonomies are the those associated with Linnaeus and his description of the Plant and Animal Kingdoms. (*https://en.wikipedia.org/wiki/File:Carl_von_Linn%C3%A9.jpg*) Today they may not be considered the most scientifically accurate depictions of nature, but they certainly have been useful over the centuries in helping humans think about nature.

There are, however, other useful ways to structure our thinking about data sets. The taxonomy is good at showing hierarchical relationships among nodes in a data set, but it's not very good at showing the properties or attributes of individual nodes. For that we're more likely to turn to a two-dimensional, matrix-like structure in which each row represents the properties of an individual node, while each column holds the value of a particular property across all nodes. We see this in spreadsheets, in relational database tables and, should we choose to work in the plain-text world, in CSV files.

Can We Represent Hierarchically Structured Data in Row-Column Format?

Materialized Path

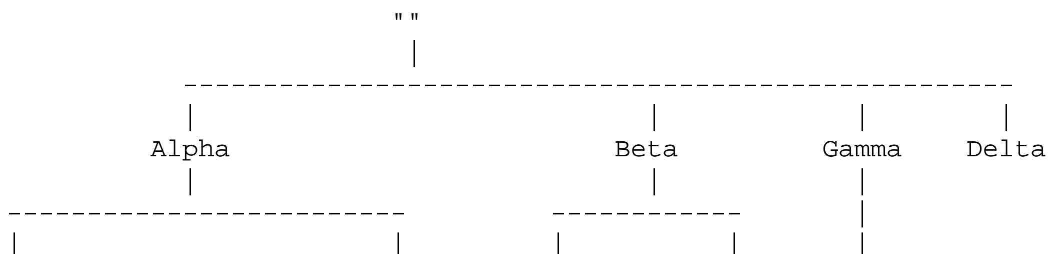
Suppose we have a set of demographic data with thse four properties:

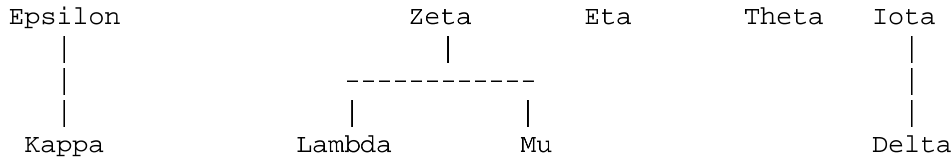
```
nationality,gender,age,income
```

With only these four columns, the data is not very useful. We don't have a way to uniquely identify a given record or look up its properties. But if the data set meets the requirements for a taxonomy we are on the road to success -- and, as we shall see, there are at least two such roads to success.

Let's go back to Diagram 3: the taxonomy where one node shared the same name with a node with a different parent:

Diagram 3: (redux)





Let's assume that each node in this tree has a value for the four demographic properties mentioned above. Our first pass at representing this tree in CSV format might look like this:

Diagram 5:

```
"name","nationality","gender","age","income"
"Alpha","","","",""
"Epsilon","","","",""
"Kappa","","","",""
"Zeta","","","",""
"Lambda","","","",""
"Mu","","","",""
"Beta","","","",""
"Eta","","","",""
"Theta","","","",""
"Gamma","","","",""
"Iota","","","",""
"Delta","","","",""
"Delta","","","",""
```

We simply create a column called `name` and populate it with the name of the node in the tree. But this is obviously insufficient: All sense of the hierarchical structure is lost; and two nodes share the same name, precluding the `name` column from being used to uniquely identify each record.

But, what if instead of a `name` column we created a `path` column which we populated, not just with the node's name, but with the names of each of the node's ancestors? In other words, suppose the `path` column held the complete route from the root node to any given node. We could do this if we separated each of the ancestor nodes with a delimiter **different** from that used in the CSV data itself. For example, if we used a pipe (`|`) delimiter inside the `path` column, our data would look like this:

Diagram 6: Taxonomy by Materialized Path

```
"path","nationality","gender","age","income"
"|Alpha","","","",""
"|Alpha|Epsilon","","","",""
"|Alpha|Epsilon|Kappa","","","",""
"|Alpha|Zeta","","","",""
"|Alpha|Zeta|Lambda","","","",""
"|Alpha|Zeta|Mu","","","",""
"|Beta","","","",""
"|Beta|Eta","","","",""
"|Beta|Theta","","","",""
"|Gamma","","","",""
"|Gamma|Iota","","","",""
"|Gamma|Iota|Delta","","","",""
"|Delta","","","",""
```

The `path` column is now an instance of what is often referred to as a **materialized path**, so I will refer to this type of CSV data as a **taxonomy by materialized path**.

Note that every node in the hierarchy (except the root node, which can often be treated as optional) has its own CSV record. Note further that the materialized path uniquely identifies each record and describes each record's position in the taxonomy.

Adjacent List

There are, however, other ways to represent a taxonomy in CSV format besides materialized path. Suppose we were to start from our last diagram and issue each record a serial number -- a unique, numerical ID issued in sequence.

Diagram 7:

```
id,"path","nationality","gender","age","income"
1,"|Alpha","","","",""
2,"|Alpha|Epsilon","","","",""
3,"|Alpha|Epsilon|Kappa","","","",""
4,"|Alpha|Zeta","","","",""
5,"|Alpha|Zeta|Lambda","","","",""
6,"|Alpha|Zeta|Mu","","","",""
7,"|Beta","","","",""
8,"|Beta|Eta","","","",""
9,"|Beta|Theta","","","",""
10,"|Gamma","","","",""
11,"|Gamma|Iota","","","",""
12,"|Gamma|Iota|Delta","","","",""
13,"|Delta","","","",""
```

Note that we now have **two** columns in this CSV file each of which uniquely identifies each record.

Next, suppose that we provide each record with a `parent_id` value, *i.e.,* the value of the `id` field for the record's parent node in the hierarchy. Our taxonomy would then look like this:

Diagram 8:

```
id,parent_id,"path","nationality","gender","age","income"
1,,"|Alpha","","","",""
2,1,"|Alpha|Epsilon","","","",""
3,2,"|Alpha|Epsilon|Kappa","","","",""
4,1,"|Alpha|Zeta","","","",""
5,4,"|Alpha|Zeta|Lambda","","","",""
6,4,"|Alpha|Zeta|Mu","","","",""
7,,"|Beta","","","",""
8,7,"|Beta|Eta","","","",""
9,7,"|Beta|Theta","","","",""
10,,"|Gamma","","","",""
11,10,"|Gamma|Iota","","","",""
12,11,"|Gamma|Iota|Delta","","","",""
13,,"|Delta","","","",""
```

Records `1`, `7`, `10` and `13` have a null value for `parent_id` because their parent is the root node (which we continue to ignore for the time being). Records `2` and `4` have a `parent_id` value of `1` because `Epsilon` and `Zeta` are the children of `Alpha` and so forth.

But there's one more refinement we need to make. We no longer need all the data in the `path` column. We can look up each node's parent by checking the value of its `parent_id` column; look up **that** node's parent by checking the value of **its** `parent_id` column; and so on until we get to the top-level node. We can revert to having a `name` column which holds only the name of an individual node. Our CSV data now looks like this:

Diagram 9:

```
id,parent_id,"name","nationality","gender","age","income"
1,,"Alpha","","","",""
2,1,"Epsilon","","","",""
3,2,"Kappa","","","",""
4,1,"Zeta","","","",""
5,4,"Lambda","","","",""
6,4,"Mu","","","",""
7,,"Beta","","","",""
8,7,"Eta","","","",""
9,7,"Theta","","","",""
10,,"Gamma","","","",""
11,10,"Iota","","","",""
12,11,"Delta","","","",""
13,,"Delta","","","",""
```

We no longer have to worry about records at different levels of the hierarchy sharing the same name because the uniqueness of each record is guaranteed by the `id` column -- not by the `name` column or by a `path` column. And we no longer need the `path` column to hold a delimited string representation of the route from the node back to the root node; we can follow the chain of `id`s and `parent_id`s to achieve the same end.

A fancier way of asking "Where is this node in the hierarchy?" is to ask "What node is this node **adjacent to**?" The term **adjacent list** is often used to describe this kind of data structure. So we'll describe this kind of taxonomy where you use `id`s and `parent_id`s to uniquely identify a node and to describe its location in the tree as a **taxonomy by adjacent list**. Note that the same underlying data structure could be represented by either a taxonomy-by-materialized-path or a taxonomy-by-adjacent-list and that we have, in effect, transformed the former into the latter.

Why Should We Care about Taxonomies?

At this point in this presentation, you are probably asking yourself two questions:

1 What's the practical use of this stuff?

2 Where's the Perl?

Let's look at some use-cases first. I'll construct these use cases from the digital marketing industry (because that's where I'm currently employed), but I'm sure you'll be able to think of other cases as well.

Digital marketing is basically worthless unless it's targeted to particular potential customers. A sub-industry of **audience vendors** has arisen whose members claim to be able to deliver particular groups of potential customers to advertisers willing to pay a fee for that service. **Advertising technology companies** need to be able to get data from audience vendors, store it internally and enable their customers -- advertising agencies and advertising brands -- to use that data in making decisions about purchasing digital advertising placements.

Let's suppose that a particular audience vendor specializes in automobiles and can provide audiences based on manufacturer, brand and model. Let's suppose further that that vendor supplies its data to ad

tech companies in a CSV file said to hold a taxonomy by materialized path. That file might look like this:

Diagram 10:

```
$> cat automobiles_taxonomy.csv

path,audience_size,cpm,buyable
"|General Motors","","",0
"|General Motors|Chevrolet",1000000,0.80,1
"|General Motors|Chevrolet|Spark",100000,0.15,1
"|General Motors|Chevrolet|Sonic",150000,0.20,1
"|General Motors|Chevrolet|Cruze",250000,0.20,1
"|General Motors|Chevrolet|Malibu",200000,0.25,1
"|General Motors|Chevrolet|Impala",175000,0.30,1
"|General Motors|Cadillac",150000,1.00,1
"|General Motors|Cadillac|Coupe",150000,0.30,1
"|General Motors|Cadillac|Sedan",150000,0.35,1
"|General Motors|Cadillac|Crossover",150000,0.40,1
"|General Motors|Cadillac|Escalade",150000,0.50,1
"|Toyota","","",0
"|Toyota|Corolla",800000,0.60,1
"|Toyota|Camry",600000,0.70,1
"|Toyota|Prius",400000,0.80,1
"|Toyota|RAV4",300000,0.50,1
```

Problem: Is the taxonomy valid?

Let's say that you're a product manager or an engineer in an ad tech company and an audience vendor has provided you with the CSV data above. Before you go inserting this data into your database and permitting your customers to make purchasing decisions based on this data, you should ask: "Is this a valid taxonomy? Can I rule out obvious defects in the data?"

Two potential defects come to mind.

1 Missing branch node

Suppose the GM data in the taxonomy above jumped directly from the `|General Motors` node to the `|General Motors|Chevrolet|Spark` node without the intermediate branch node `|General Motors|Chevrolet`.

```
"|General Motors","","",0
"|General Motors|Chevrolet|Spark",100000,0.15,1
```

This would violate the rule that every node in the hierarchy be represented by a unique record in the taxonomy. Hence, the taxonomy would be invalid.

2 Two nodes, children of same parent, with same name

Suppose in the `Chevrolet` data there were two nodes with the same brand name.

```
"|General Motors|Chevrolet|Sonic",100000,0.15,1
"|General Motors|Chevrolet|Sonic",150000,0.20,1
```

Two children of the same parent node are sharing the same name. This would invalidate the taxonomy because the materialized path would no longer be able to uniquely identify each record.

Solution: Parse::Taxonomy

I have written a Perl library called *Parse-Taxonomy* which you can find on CPAN which provides basic functionality for handling taxonomies with data in CSV format. The library contains two principal subclasses: *Parse::Taxonomy::MaterializedPath* and *Parse::Taxonomy::AdjacentList* to handle the two kinds of taxonomies discussed in today's presentation. The library also contains a **cookbook** with **recipes** for common problems in dealing with taxonomies. Consulting that cookbook, we find that we can validate a taxonomy-by-materialized-path by simply trying to create a *Parse::Taxonomy::MaterializedPath* object.

```
$obj = Parse::Taxonomy::MaterializedPath->new( {
    file => 'automobiles_taxonomy.csv',
} );
ok(defined $obj, "$taxonomy_file passes basic validations");
```

Once we have a valid taxonomy, we can play with it by calling methods on the object. For example, we can turn it into a hash:

```
my $hashified = $obj->hashify();

{
    "|General Motors" => {
        audience_size => "",
        buyable => 0,
        cpm => "",
        path => "|General Motors"
    },
    "|General Motors|Cadillac" => {
        audience_size => 150000,
        buyable => 1,
        cpm => "1.00",
        path => "|General Motors|Cadillac",
    },
    # ...
    "|Toyota" => {
        audience_size => "",
        buyable => 0,
        cpm => "",
        path => "|Toyota"
    },
    "|Toyota|Camry" => {
        audience_size => 600000,
        buyablB => 1,
        cpm => "0.70",
        path => "|Toyota|Camry",
    },
    # ...
}
```

We can also take the data in the object and structure them as a taxonomy-by-adjacent-list, which we then write out to a CSV file.

```
my $adjacentified = $obj->adjacentify();
```

```
my $csv_out = "$cwd/examples/data/auto_adjacent.csv";

$obj->write_adjacentified_to_csv( {
    adjacentified => $adjacentified,
    csvfile => $csv_out,
} );

$> cat examples/data/auto_adjacent.csv

id,parent_id,name,audience_size,cpm,buyable
1,,"General Motors",,,0
2,,Toyota,,,0
3,1,Chevrolet,1000000,0.80,1
4,1,Cadillac,150000,1.00,1
5,2,Corolla,800000,0.60,1
6,2,Camry,600000,0.70,1
7,2,Prius,400000,0.80,1
8,2,RAV4,300000,0.50,1
9,3,Spark,100000,0.15,1
10,3,Sonic,150000,0.20,1
11,3,Cruze,250000,0.20,1
12,3,Malibu,200000,0.25,1
13,3,Impala,175000,0.30,1
14,4,Coupe,150000,0.30,1
15,4,Sedan,150000,0.35,1
16,4,Crossover,150000,0.40,1
17,4,Escalade,150000,0.50,1
```

We could next take this CSV file and use it as input for another module within the *Parse-Taxonomy* distribution, *Parse::Taxonomy::AdjacentList*. As was the case with *Parse::Taxonomy::MaterializedPath*, if we can successfully create a *Parse::Taxonomy::AdjacentList* object with a given input, we have confirmed that the data has a taxonomically valid structure.

```
my $alobj = Parse::Taxonomy::AdjacentList->new( {
    file => $csv_out,
} );
ok(defined($alobj), "Taxonomically valid adjacent list");
```

We can even turn this taxonomy-by-adjacent-list back into a taxonomy-by-materialized-path by calling a `pathify()` method on this object.

```
my $pathified = $alobj->pathify();
```

And, with a little work not shown here, we can write tests that demonstrate that the data structure in the Parse::Taxonomy::AdjacentList object is exactly the same as the data structure we first created in the Parse::Taxonomy::MaterializedPath object.

Summary

In today's presentation we've:

- Met the tree-like data structure known as a taxonomy;

- Seen how taxonomies can be represented in a two-dimensional, row-column matrix format such as a CSV file; and
- Examined how CPAN library *Parse-Taxonomy* can be used to validate a data structure as a taxonomy and to transform one common form of taxonomy into another.

There are a variety of other CPAN libraries which deal with hierarchically structured data, particularly in the DBIx::Class namespace. I encourage you to check them all out.

Thank you very much.