

Re: Pyparsing help

Source: <http://coding.derkeiler.com/Archive/Python/comp.lang.python/2008-03/msg02792.html>

- *From:* Paul McGuire <ptmcg@xxxxxxxxxxxxxxxx>
 - *Date:* Sun, 23 Mar 2008 00:26:36 -0700 (PDT)
-

There are a couple of bugs in our program so far.

First of all, our grammar isn't parsing the METAL2 entry at all. We should change this line:

```
md = mainDict.parseString(test1)
```

to

```
md = (mainDict+stringEnd).parseString(test1)
```

The parser is reading as far as it can, but then stopping once successful parsing is no longer possible. Since there is at least one valid entry matching the OneOrMore expression, then parseString raises no errors. By adding "+stringEnd" to our expression to be parsed, we are saying "once parsing is finished, we should be at the end of the input string". By making this change, we now get this parse exception:

```
pyparsing.ParseException: Expected stringEnd (at char 1948), (line:54, col:1)
```

So what is the matter with the METAL2 entries? After using brute force "divide and conquer" (I deleted half of the entries and got a successful parse, then restored half of the entries I removed, until I added back the entry that caused the parse to fail), I found these lines in the input:

```
fatTblThreshold = (0,0.39,10.005)  
fatTblParallelLength = (0,1,0)
```

Both of these violate the atflist definition, because they contain integers, not just floatnums. So we need to expand the definition of atflist:

```
floatnum = Combine(Word(nums) + "." + Word(nums) +  
Optional('e'+oneOf("+ -")+Word(nums)))  
floatnum.setParseAction(lambda t:float(t[0]))  
integer = Word(nums).setParseAction(lambda t:int(t[0]))
```

Re: Pyparsing help

```
atflist = Suppress("(") + delimitedList(floatnum|integer) + \  
Suppress(")")
```

Then we need to tackle the issue of adding nesting for those entries that have sub-keys. This is actually kind of tricky for your data example, because nesting within Dict expects input data to be nested. That is, nesting Dict's is normally done with data that is input like:

```
main  
Technology  
Layer  
PRBOUNDARY  
METAL2  
Tile  
unit
```

But your data is structured slightly differently:

```
main  
Technology  
Layer PRBOUNDARY  
Layer METAL2  
Tile unit
```

Because Layer is repeated, the second entry creates a new node named "Layer" at the second level, and the first "Layer" entry is lost. To fix this, we need to combine Layer and the layer id into a composite-type of key. I did this by using Group, and adding the Optional alias (which I see now is a poor name, "layerId" would be better) as a second element of the key:

```
mainDict = dictOf(  
Group(Word(alphas)+Optional(quotedString)),  
Suppress("{}" + attrDict + Suppress("{}")  
)
```

But now if we parse the input with this mainDict, we see that the keys are no longer nice simple strings, but they are 1- or 2-element ParseResults objects. Here is what I get from the command "print md.keys()":

```
[(['Technology'], {}), (['Tile', 'unit'], {}), (['Layer',  
'PRBOUNDARY'], {}), (['Layer', 'METAL2'], {})]
```

So to finally clear this up, we need one more parse action, attached to the mainDict expression, that rearranges the subdicts using the elements in the keys. The parse action looks like this, and it will process the overall parse results for the entire data structure:

```
def rearrangeSubDicts(toks):  
# iterate over all key-value pairs in the dict
```

Re: Pyparsing help

Re: Pyparsing help

```
for key,value in toks.items():
# key is of the form ['name'] or ['name', 'name2']
# and the value is the attrDict

# if key has just one element, use it to define
# a simple string key
if len(key)==1:
toks[key[0]] = value
else:
# if the key has two elements, create a
# subnode with the first element
if key[0] not in toks:
toks[key[0]] = ParseResults([])

# add an entry for the second key element
toks[key[0]][key[1]] = value

# now delete the original key that is the form
# ['name'] or ['name', 'name2']
del toks[key]
```

It looks a bit messy, but the point is to modify the tokens in place, by rearranging the attrdicts to nodes with simple string keys, instead of keys nested in structures.

Lastly, we attach the parse action in the usual way:

```
mainDict.setParseAction(rearrangeSubDicts)
```

Now you can access the fields of the different layers as:

```
print md.Layer.METAL2.lineStyle
```

I guess this all looks pretty convoluted. You might be better off just doing your own Group'ing, and then navigating the nested lists to build your own dict or other data structure.

— Paul

.