PARSEARG: TURNS ARGPARSE ON ITS HEAD, THE DECLARATIVE WAY

THOMAS P. HARTE

CONTENTS

1. QUICKSTART

1.1. **Overview.** `parsearg` is a Python package for writing command-line interfaces ("CLI") that augments (rather than replaces) the standard Python module for writing CLIs, `argparse`. There is nothing wrong with `argparse`: It's fine in terms of the *functionality* that it provides, but it can be clunky to use, especially when a program's structure has subcommands, or nested subcommands (*i.e.* subcommands that have subcommands). Moreover, because of the imperative nature of `argparse`, it makes it hard to understand how a program's interface is structured (*viz.* the program's "view").

`parsearg` puts a layer on top of `argparse` that makes writing a CLI easy: You declare your view (*i.e.* the CLI), with a `dict` so that the view is a data structure (*i.e.* pure configuration). The data structure declares the *intent* of the CLI and you no longer have to instruct `argparse` on how to put the CLI together: `parsearg` does that for you. In this respect, `parsearg` turns `argparse` on its head, in the sense that it replaces imperative instructions with declarative data.

1.2. **Usage.** Suppose we wish to create a program called `quickstart-todos.py` to manage the TO-DOs of a set of different users. We want to have subprograms of `quickstart-todos.py`; for example, we may want to create a user (`python quickstart-todos.py create user`, say), or we may want to create a TO-DO for a particular user (`python quickstart-todos.py create todo`, say). We might also want to add optional parameters to each subprogram such as the user's email and phone number, or the TO-DO's due date. An invocation of the program's CLI might look like the following:

```
python quickstart-todos.py create user Bob --email=bob@email.com --phone=+1-212-555-1234
python quickstart-todos.py create todo Bob 'taxes' --due-date=2021-05-17
```

Date: 2022-05-31 Build: `v-0.3.6 (c315a0952449037e963c95c8c0de800d0a5548d3)`.

With `argparse`, the subprogram `create` would necessitate fiddling with subparsers. With `parsearg`, the CLI for the above is declared with a `dict` and `parsearg.parser.ParseArg` supplants the normal use of `argparse.ArgumentParser`. Moreover, the callback associated with each subcommand is explicitly linked to its declaration.

```python
import sys
from parsearg import ParseArg

def create_user(args):
    print(f'created user: {args.name!r} (email: {args.email}, phone: {args.phone})')

def create_todo(args):
    print(f'created TO-DO for user {args.user!r}: {args.title} (due: {args.due_date})')

view = {
    'create|user': {
        'callback':   create_user,
        'name':       {'help': 'create user name', 'action': 'store'},
        '-e|--email': {'help': "create user's email address", 'action': 'store', 'default': ''},
        '-p|--phone': {'help': "create user's phone number", 'action': 'store', 'default': ''},
    },
    'create|todo': {
        'callback':   create_todo,
        'user':       {'help': 'user name', 'action': 'store'},
        'title':      {'help': 'title of TO-DO', 'action': 'store'},
        '-d|--due-date': {'help': 'due date for the TO-DO', 'action': 'store', 'default': None},
    },
}

def main(args):
    # ParseArg takes the place of argparse.ArgumentParser
    parser = ParseArg(d=view)

    # parser.parse_args returns an argparse.Namespace
    ns     = parser.parse_args(args)

    # ns.callback contains the function in the 'callback' key of 'view'
    result = ns.callback(ns)

if __name__ == "__main__":
    args = sys.argv[1:] if len(sys.argv) > 1 else []

    main(' '.join(args))
```

A fully-worked version of the TO-DO example is presented in the docs. The output of the above is:

```
python quickstart-todos.py create user Bob --email=bob@email.com --phone=212-555-1234
```

```
created user: 'Bob' (email: bob@email.com, phone: 212-555-1234)
```

```
python quickstart-todos.py create todo Bob 'taxes' --due-date=2021-05-17
```

```
created TO-DO for user 'Bob': taxes (due: 2021-05-17)
```

Because `parsearg` is built on top of `argparse`, all the usual features are available, such as the extensive help features (essentially making the CLI self-documenting):

```
python quickstart-todos.py --help
```

```
usage: quickstart-todos.py [-h] {create} ...

positional arguments:
  {create}
```

```
optional arguments:
  -h, --help  show this help message and exit
```

```
python quickstart-todos.py create --help
```

```
usage: quickstart-todos.py create [-h] {user,todo} ...

positional arguments:
  {user,todo}

optional arguments:
  -h, --help   show this help message and exit
```

```
python quickstart-todos.py create user --help
```

```
usage: quickstart-todos.py create user [-h] [-e EMAIL] [-p PHONE] name

positional arguments:
  name                  create user name

optional arguments:
  -h, --help            show this help message and exit
  -e EMAIL, --email EMAIL
                        create user's email address
  -p PHONE, --phone PHONE
                        create user's phone number
```

```
python quickstart-todos.py create todo --help
```

```
usage: quickstart-todos.py create todo [-h] [-d DUE_DATE] user title

positional arguments:
  user                  user name
  title                 title of TO-DO

optional arguments:
  -h, --help            show this help message and exit
  -d DUE_DATE, --due-date DUE_DATE
                        due date for the TO-DO
```

2. Overview

The "standard" Python module for writing command-line interfaces ("CLI") is `argparse`. It is standard in so far as it is one of the batteries that comes included with the Python distribution, so no special installation is required. Probably because `argparse` is a bit clunky to use, many other (non-standard) packages have been developed for creating CLIs. Why "clunky"? Putting together a CLI with `argparse` alone is nothing if not an exercise in imperative programming, and this has negative consequences:

(1) It obfuscates the intention of the CLI design;

(2) It is prone to errors;

(3) It discourages CLI design in the first instance;

(4) It makes debugging a CLI design very difficult; and

(5) It makes refactoring or re-configuring the CLI design overly burdensome.

In spite of this clunkiness, `argparse` has everything we need in terms of functionality. `parsearg`, then, is nothing more than a layer over `argparse` that exposes the `argparse` functionality via a `dict`. The `dict` is the View component of the Model-View-Controller ("MVC") design pattern. The `dict` embeds callbacks from the Controller component, thereby achieving a clean separation of duties, which is what the MVC pattern calls for. By separating the View component into a `dict`, the CLI design can be expressed in a declarative way: `parsearg` manifests the *intention* of the CLI design without having to specify how that design is implemented in terms of `argparse`'s parsers and subparsers (`parsearg` does that for you).

Other packages—such as `click` and `plac`—effectively decorate functions that are part of the Controller with functionality from the View. Unfortunately, while this may expose the functionality of `argparse` in a more friendly way via the packages' decorators, it dissipates the elements of the View across the Controller and in so doing it makes the CLI design difficult to grasp.

The `parsearg` philosophy is that `argparse` is already good enough in terms of the functionality that it provides, but that it just needs a little nudge in terms of how it's used. Arguments to be added to a CLI with `argparse` can be clearly specified as data, as can the callbacks that consume these arguments. `parsearg` takes advantage of this by specifiying everything (in the View component of MVC) as a `dict`, from which `parsearg` then generates a parser (or set of nested parsers) using `argparse`. The Controller is then free to use the generated parser.

The `parsearg` approach is declarative because it manifests the CLI design in a data structure: a `dict`, which is one of Python's built-in data structures. The keys of this `dict` form a flattened tree of the CLI's subcommands. Keys like `A|B|C` are easy to specify and neatly summarize the nested hierarchy of subcommands: `A -> B -> C`. They are also easy to change. The magic, such as it is, of `parsearg` is that it unflattens this flattened tree into a tree of `argparse` parsers. `parsearg` requires nothing special: It works with Python out of the box, and therefore uses what's already available without introducing dependencies.

Simple? The following examples should help.

3. How PARSEARG works

3.1. **Heterogeneous trees: recursive data structures.**

Trees are recursive data structures. Any node of a tree is identified by a value and a set of *child trees*, thus establishing the recursion.

Definition 3.1. *An heterogeneous tree is either:*

(1) a value and a set of child trees; or,

(2) the empty tree.

Remark 3.1. *The value is also known as a "node", or "name"; its set of child trees (also known as the "children" of the value/node/name) is possibly empty.*

Remark 3.2. *The value/node/name is referred to as the "parent" of its children.*

Remark 3.3. *The children of a unique parent tree are termed "siblings", though they are not necessarily aware of each other.*

Remark 3.4. *The empty tree has no value/node/name and an empty set of children.*

Remark 3.5. *The definition implies that a parent tree "points" to its children, in so far as the parent is necessarily aware of the children it contains, whereas its children are not necessarily aware of their parent, nor of each other. We will therefore regard Definition 3.1 as the "top-down" definition.*

In Python, we represent the value/node/name and its `set` of children as the `tuple`:

```
(value, {e1, e2, ....}),
```

where `{e1, e2, ...}` is the `set` of children containing elements `e1`, `e2`, and so on, and each element of the `set` is itself a tree. The empty tree is represented by

```
(None, {None}),
```

where `None` represents no value, and `{None}` is the empty set of children.[1]

QUESTION:

> Is `(value, {(None, {})})` equivalent to `(value, {})`?
>
> This would mean that the representation of an empty tree, *viz.* `(None, {})`, is perfectly equivalent to the null set $\emptyset$, *i.e.* `(value, {(None, {})})` is equivalent to `(value,` $\{\emptyset\}$`)` is equivalent to `(value, {})`. But this requires a little mental gymnastics: Isn't the *tuple* `(None, {})` equivalent to $(\emptyset, \{\emptyset\})$, which is *not* the same as $\{\emptyset\} \equiv$ `{}`, *i.e.* a *tuple* containing a null value and the null set is not the same as the null set itself?

This type of tree is termed *heterogeneous* because the cardinality of the set of child trees for each value can vary. Contrast with an n-ary tree, which has the same number of children at each node (*e.g.* two for a binary tree, three for a ternary tree, and in general n for an an n-ary tree).

Nothing in the definition precludes the possibility of a child tree's value from containing an identical value to that of its parent. Thus,

```
(value, {e1, (value, {e1})),
```

is perfectly valid. The requirement does explicitly require each of node's child trees to be unique, in so far as they are members of a *set*.

---

[1] Python has a quirk where `{}` is of type `dict`, whereas `{None}` is of type `set`. Thus, the "empty set" is properly represented by `{None}` and not `{}`.

Definition 3.2. *A terminating (or "leaf") node is:*

> *a tree whose set of child trees is empty.*

A leaf node is represented by `(value, {})`. Note that the empty tree, `(None, {})`, is also a valid leaf node.

We can define an heterogeneous tree in reverse, *viz.* instead of starting with a parent and recursively defining the tree through its children, we can start with a terminating node and recursively define the tree by identifying each node's parent.

This definition yields the flip side of the "top-down" definition, above.

Definition 3.3. *An heterogeneous tree is either:*

(1) a set of sibling trees sharing a common parent value/node/name; or,

(2) the empty tree.

The set of sibling trees that shares a common parent is possibly empty. What is the parent of an empty set (*i.e.* that contains no trees)? It is the same as any other tree: It must point to its parent. Optionally, siblings may point to each other. Further still, a child node may point to all of its ancestors (its parent's parent, and so on).

In Python, we represent the "reverse tree" as.

TODO:

> Insert appropriate representation here.

Remark 3.6. *Definition 3.3 implies that a child tree points to its parent, which means that the child must be able to uniquely identify the parent tree in which it is contained. Because a parent contains its children, a parent must be aware of its children, so it is not strictly the reverse of the "top-down" definition. Nonetheless, we'll refer to Definition 3.3 as the "bottom-up" definition.*

3.2. **Trees: flat and unflattened.** While it's possible to create a "longhand" version of the A tree using the `Tree` class that comes with the `parsearg` package, it is cumbersome to use and—more specifically—it can easily obfuscate the tree structure itself when the node names and payloads are in any way complex.[2]

```python
"""
       A
     / | \
    B  BB BBB
      /| \
     C CC CCC
"""

from parsearg.data_structures import Tree, Node, Key

tree = Tree('A', children=[
    Tree('B', []),
    Tree('BB', children=[
        Tree('C', []),
        Tree('CC', []),
        Tree('CCC', [])
    ]),
    Tree('BBB', []),
])

print(tree.show(quiet=True))
```

[2]For clarity the nodes' payloads, *i.e.* the nodes' values, are omitted here.

```
A
    B
    BB
        C
        CC
        CCC
    BBB
```

We need a "shorthand" for constructing trees. To this end, `parsearg` uses a set of pipe-delimited strings:

```python
# from a import view
from examples.a import view

print(['root'] + list(view.keys()))
```

```
['root', 'A', 'A|B', 'A|BB', 'A|BB|C', 'A|BB|CC', 'A|BB|CCC', 'A|BBB']
```

This representation can be considered to be a "flattened" version of the A Tree:

```python
print(
    ParseArg.to_tree(view, root_name='root').show(quiet=True)
)
```

```
root
    A
        BB
            CCC
            C
            CC
        BBB
        B
```

Each string in this "flat tree" (or "flattened tree") represents a node in the tree and uniquely names that node. The singular constraint on the tree's structure is that each node's children must contain a unique set of node names. If this were not the case, then the tree's structure would be indeterminate. In point of fact, the example of the A-AA Tree illustrates that the structure of the A Tree can be duplicated: only the root node names of each sub-tree (`A` and `AA`, respectively) need to be unique.

Leaf nodes are terminating nodes in the tree's hierarchy (*i.e.* nodes that have no children). In the above, the terminating nodes are easily discerned visually within the tree's hierarchy:

```python
print( ParseArg.to_tree(view, root_name='root').show(quiet=True) )
```

```
root
    A
        BB
            CCC
            C
            CC
        BBB
        B
```

Thus, the terminating nodes are:

```
'A|B'
'A|BBB'
'A|BB|C'
'A|BB|CC'
'A|BB|CCC'
```

while the non-terminating nodes are:

```
'A'
'A|BB'
```

The separator in each key (here, a pipe `|`) separates one level of the tree from the next level down. It is this delimited-string representation that allows us to easily specify a tree. However, taking advantage of a tree as a data structure requires working with a tree, not a flattened version of it. In other words, we have a flattened-tree which is shorthand to specify the tree, but we need to unflatten this shorthand to create the tree data structure in order for it to be useful.

There is a duality here:

(1) we need a shorthand in order for the tree's structure to be specified in a simple way (the "flattened" shorthand); and

(2) we need to make use of the structure of the tree itself (the "unflattened" tree).

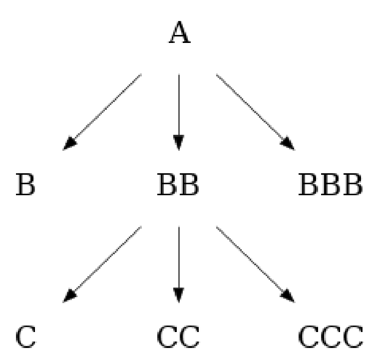
Thus, `parsearg` transforms a `dict`, representing the tree's nodes, into a `Tree`. The keys of the `dict` are shorthand for the tree's nodes and the values of the `dict` are the nodes' payloads. We need the shorthand of the `dict` for ease of specification, and we need the `Tree` to avail of the data structure's properties.

3.3. **The A Tree and the AA Tree.** In order to further explain how `parsearg` works, we will make repeated use two simple abstractions, based on nested trees (or "sub-trees"):

(1) The "A Tree", and

(2) The "A-AA Tree".

3.3.1. *The A Tree.* We have already seen the A Tree. It has three levels.

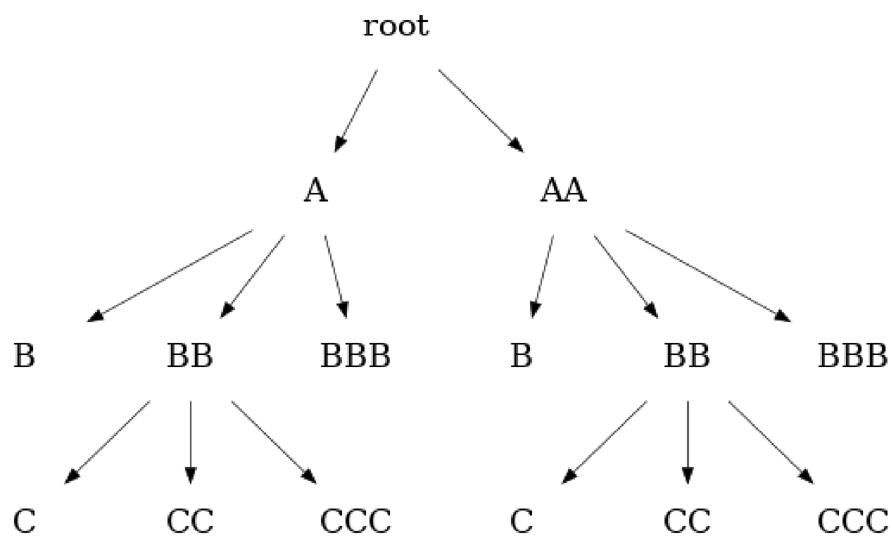


Let's take a very brief look at the Python code for `a.py`. Consider the `dict` that represents the View component. The tree that the parsed `dict` is represented by is given as:

```
from examples.a import view

print(
    ParseArg(d=view, root_name='root').tree.show(quiet=True)
)
```

```
root
    A
        BB
            CCC
            C
            CC
        BBB
        B
```

3.3.2. *The A-AA Tree.* The A-AA Tree simply extends the A Tree one level and replicates the A Tree's structure in a sub-tree with root name "AA":



Let's take a brief look at the Python code for `a_aa.py`. Consider the `dict` that represents the View component: and then the tree that the parsed `dict` is represented by:

```python
from examples.a_aa import view

print(
    ParseArg(d=view, root_name='TODO').tree.show(quiet=True)
)
```

```
TODO
    A
        BB
            CCC
            C
            CC
        BBB
        B
    AA
        BB
            CCC
            C
            CC
        BBB
        B
```

3.4. **Some useful helper data structures.** `parsearg` provides three core classes:

```python
from parsearg.data_structures import Key, Node, Tree
```

Both `Key` and `Node` help to marshall keys and their payloads when unflattening a flat tree structure. The `Tree` class represents the actual tree.

`Key` and `Node` commandeer the data in the `dict` in the View layer. Their principle use is in unflattening the View `dict` by left-shifting the keys that represent the nodes of the tree. The basic algorithm for rendering an unflattened tree from a flat tree follows the logic of splitting the key (*i.e.* the `str` representing the key)

```python
# 1. take key
key = 'A|B|C'
```

```
# 2. split string
sep = '|'
print( Key.split(key) )
```

```
['A', '|', 'B', '|', 'C']
```

```
# 3. unflatten the split key (i.e. create a nested list)
unflattened = Key.unflatten(Key.split(key))
print(unflattened)
```

```
['A', ['B', ['C']]]
```

```
# 4. create a Node from the unflattened list
node = Node.from_nested_list(unflattened)
print(node)
```

```
('A', ['B', ['C']])
```

```
# 5. the node can now be shifted until the empty node is reached
print(node << 1)
print(node << 2)
print(node << 3)
```

```
('B', ['C'])
('C', [])
(None, [])
```

Note that the `key`, above, can be directly transformed into a `Node`:

```
print( key )
print( Key.to_node(key) )
```

```
A|B|C
('A', ['B', ['C']])
```

A `Node` is really a convenient way to view the `key`. Shifting the `Node` until it is empty provides a convenient way to determine when a leaf node has been reached:

```
n = node
i = 0
while not n.is_empty():
    i += 1
    n = node << i

print(i)
print(node << i)
```

```
3
(None, [])
```

A `Key` just keeps track of the node's payload and the key itself while this iteration is performed over all nodes at a particular level of the flat tree.

3.4.1. *The `Key` class.* First, `parsearg` defines a `Key` class. Take a key `'A|BB|C'` that represents a leaf node:

```
from parsearg.data_structures import Tree, Node, Key

key = 'A|BB|C'
print( Key(key) )
```

```
'A|BB|C': ('A', ['BB', ['C']])
```

The `Key` stores the following:

(1) the key:

```
print( Key(key).key )
```

```
A|BB|C
```

(2) the `dict` assocated with the key (if any):

```
payload = 3.141593
d = {key: payload}

print( Key(key, d=d).d )
```

```
{'A|BB|C': 3.141593}
```

(3) the `Node`, which is a representation of the key's delimited string as a nested list:

```
print( Key(key).value )
print( type(Key(key).value) )
```

```
('A', ['BB', ['C']])
<class 'parsearg.data_structures.Node'>
```

(4) the `payload` (*i.e.* the value of the `dict`, as above):

```
payload = 3.141593
d = {key: payload}

print( Key(key, d).payload )
```

```
3.141593
```

A `Key` can be shifted:

```
print( Key(key) )
# same thing:
print( Key(key) << 0 )
# real shifts:
print( Key(key) << 1 )
print( Key(key) << 2 )
print( Key(key) << 3 )
# same thing:
print( Key(key) << 4 )
```

```
'A|BB|C': ('A', ['BB', ['C']])
'A|BB|C': ('A', ['BB', ['C']])
'A|BB|C': ('BB', ['C'])
'A|BB|C': ('C', [])
'A|BB|C': (None, [])
'A|BB|C': (None, [])
```

but the `key` is preserved regardless of the shift value:

```
print( (Key(key) << 0).key )
print( (Key(key) << 1).key )
print( (Key(key) << 2).key )
print( (Key(key) << 3).key )
```

```
A|BB|C
A|BB|C
A|BB|C
A|BB|C
```

3.4.2. *The `Node` class.* A `Node` is a tuple that has a `head` and a `tail`:

(1) the `head` is the name of the node, and

(2) the `tail` is a nested list containing the remainder of the leaf node's specification.

For example:

```
print( Key(key).value )
print( Key(key).value.head() )
print( Key(key).value.tail() )
# use a Node to construct the ~value~ part:
print( Node(head='A', tail=['BB', ['C']]) )
```

```
('A', ['BB', ['C']])
A
['BB', ['C']]
('A', ['BB', ['C']])
```

A `Node`, upon which the `Key` is based, can also be shifted:

```
node = Node(head='A', tail=['B', ['C']])
print( node )
# same thing:
print( node << 0 )
print( node << 1 )
print( node << 2 )
print( node << 3 )
# same thing:
print( node << 4 )
```

```
('A', ['B', ['C']])
('A', ['B', ['C']])
('B', ['C'])
('C', [])
(None, [])
(None, [])
```

You can build a `Node` from a nested `list`:

```
ll = ['A', ['B', ['C']]]
print( Node.from_nested_list(ll) )
```

```
('A', ['B', ['C']])
```

```
print( Node.from_nested_list(['A', ['B', ['C']]]) )
print( Node.from_nested_list(['A']) )
```

```
('A', ['B', ['C']])
('A', [])
```

3.4.3. *Putting it together.* Let's say we have the following View:

```python
from parsearg.utils import show
d = {
    'A': {
        'arg1': {}, 'arg2': {}, 'arg3': {},
    },
    'A|B': {
        'arg1': {}, 'arg2': {}, 'arg3': {},
    },
    'A|B|C': {
        'arg1': {}, 'arg2': {}, 'arg3': {},
    },
}
show( d )
```

```
'A':
    'arg1':
    {}
    'arg2':
    {}
    'arg3':
    {}
'A|B':
    'arg1':
    {}
    'arg2':
    {}
    'arg3':
    {}
'A|B|C':
    'arg1':
    {}
    'arg2':
    {}
    'arg3':
    {}
```

We can gather the keys as `Key` objects:

```python
keys = list(map(lambda key: Key(key), list(d.keys())))
for key in keys: print(key)
```

```
'A': ('A', [])
'A|B': ('A', ['B'])
'A|B|C': ('A', ['B', ['C']])
```

We can gather the leaves (there's only one leaf node: `A`):

```python
leaves = list(filter(lambda key: key.is_leaf(), keys))
for leaf in leaves: print(leaf)
```

```
'A': ('A', [])
```

We can shift the keys and then gather the leaves (there's only one leaf node: `A|B`):

```python
leaves = list(filter(lambda key: (key << 1).is_leaf(), keys))
for leaf in leaves: print(leaf)
```

```
'A|B': ('A', ['B'])
```

We can shift the keys and then gather the leaves (there's only one leaf node: `A|B|C`):

```python
leaves = list(filter(lambda key: (key << 2).is_leaf(), keys))
for leaf in leaves: print(leaf)
```

```
'A|B|C': ('A', ['B', ['C']])
```

Note how the `Key` object preserves the key `A|B|C` even when the associated `Node` object is left-shifted:

```
node = leaves[0]
print( (node << 1).key )
print( (node << 2).key )
print( (node << 3).key )
print( (node << 3) )
```

```
A|B|C
A|B|C
A|B|C
'A|B|C': (None, [])
```

3.4.4. *The `Tree` class.* As illustrated above a tree can be constructed directly with the `Tree` class. For example the A Tree is:

```
tree = Tree('A', children=[
    Tree('B', []),
    Tree('BB', children=[
        Tree('C', []),
        Tree('CC', []),
        Tree('CCC', [])
    ]),
    Tree('BBB', []),
])

print( tree.show(quiet=True) )
```

```
A
    B
    BB
        C
        CC
        CCC
    BBB
```

The `Tree` class has a nested class called `Value`, which stores a node's value. The `Value` constructor checks that everything is in sync, as the following shows:

```
key = 'A|BB|C'
payload = 3.141593
d = {key: payload}

value = Tree.Value(name='C', key=key, d=d)

print( value )
print( value.key )
print( value.name )
print( value.payload )
```

```
C
A|BB|C
C
3.141593
```

3.5. **Creating parsers with `parsearg`.** `parsearg` creates parsers in two steps:

(1) transform the View `dict` into a `Tree`;

(2) create a parser—and subparsers—using the `Tree`.

There is nothing specific to parsers in the first step, so this forms a generic pattern: transforming a View into a tree. The second step is specific to creating command-line parsers: It traverses the tree structure created in the first step and uses it to create a tree structure of parsers.

Consider Step 1. Any `dict` that conforms to the shorthand notation for representing nodes (*i.e.* a flat tree) can be converted to a `Tree` using `parsearg.parser.ParseArg.to_tree`. Take the View `dict` from Section 1.2:

```
def create_user(x): return x
def create_todo(x): return x

view = {
    'create|user': {
        'callback':   create_user,
        'name':       {'help': 'create user name', 'action': 'store'},
        '-e|--email': {'help': "create user's email address", 'action': 'store', 'default': ''},
        '-p|--phone': {'help': "create user's phone number", 'action': 'store', 'default': ''},
    },
    'create|todo': {
        'callback':   create_todo,
        'user':       {'help': 'user name', 'action': 'store'},
        'title':      {'help': 'title of TO-DO', 'action': 'store'},
        '-d|--due-date': {'help': 'due date for the TO-DO', 'action': 'store', 'default': None},
    },
}


from parsearg import ParseArg

tree = ParseArg.to_tree(d=view)
tree.show()
```

```
root
    create
        user
        todo
```

Now consider Step 2. This is specific to building `argparse` subparsers:

```
args   = 'create user Harry -e harry@hogwarts.edu -p 212-456-1234'.split()

import argparse
parser = argparse.ArgumentParser(add_help=True)
ParseArg.make_subparsers(tree, parser)

# parser.parse_args returns an argparse.Namespace
print( parser.parse_args(args) )
```

```
Namespace(callback=<function create_user at 0x7f166ac21af0>, email='harry@hogwarts.edu', name='Harry', phone='212-456-1234')
```

The `ParseArg` constructor abstracts this process and wraps the calls to `argparse` methods:

```
parser = ParseArg(d=view)
print( parser.parse_args(args) )
```

```
Namespace(callback=<function create_user at 0x7f166ac21af0>, email='harry@hogwarts.edu', name='Harry', phone='212-456-1234')
```

3.6. Reverse-definition tree: Creating a tree from its leaf nodes.

TODO:

This section is incomplete.

```
from examples.a_aa import view
sep = '|'

k = list(map(lambda x: x.split(sep), view.keys()))
for e in k: print(e)
```

```
['A']
['A', 'B']
['A', 'BB']
['A', 'BB', 'C']
['A', 'BB', 'CC']
['A', 'BB', 'CCC']
['A', 'BBB']
['AA']
['AA', 'B']
['AA', 'BB']
['AA', 'BB', 'C']
['AA', 'BB', 'CC']
['AA', 'BB', 'CCC']
['AA', 'BBB']
```

```
n = list(map(len, k))

for key, depth, lkey in zip(view.keys(), n, k): print(f'({key}:\t{depth}, {lkey})')
```

```
(1, ['A'])
(2, ['A', 'B'])
(2, ['A', 'BB'])
(3, ['A', 'BB', 'C'])     # Tree('BB', [Tree('C'), Tree('CC'), Tree('CCC')])
(3, ['A', 'BB', 'CC'])
(3, ['A', 'BB', 'CCC'])
(2, ['A', 'BBB'])
(1, ['AA'])
(2, ['AA', 'B'])
(2, ['AA', 'BB'])
(3, ['AA', 'BB', 'C'])    # Tree('AA', [Tree('BB', [Tree('C'), Tree('CC'), Tree('CCC')])])
(3, ['AA', 'BB', 'CC'])
(3, ['AA', 'BB', 'CCC'])
(2, ['AA', 'BBB'])
```

```
(1, ['A'])
(2, ['A', 'B'])
(2, ['A', 'BB'])
(3, ['A', 'BB', 'C'])     # Tree('AA', [Tree('BB', [Tree('C'), Tree('CC'), Tree('CCC')])])
(3, ['A', 'BB', 'CC'])
(3, ['A', 'BB', 'CCC'])
(2, ['A', 'BBB'])
(1, ['AA'])
(2, ['AA', 'B'])
(2, ['AA', 'BB'])
(3, ['AA', 'BB', 'C'])    # Tree('AA', [Tree('BB', [Tree('C'), Tree('CC'), Tree('CCC')])])
(3, ['AA', 'BB', 'CC'])
(3, ['AA', 'BB', 'CCC'])
(2, ['AA', 'BBB'])


# Tree('A',  [Tree('BB', [Tree('C'), Tree('CC'), Tree('CCC')])])
# Tree('AA', [Tree('BB', [Tree('C'), Tree('CC'), Tree('CCC')])])
(1, ['A'])
(2, ['A', 'B'])
(2, ['A', 'BB'])
(2, ['A', 'BBB'])
(1, ['AA'])
(2, ['AA', 'B'])
(2, ['AA', 'BB'])
(2, ['AA', 'BBB'])
```

```
import pandas as pd
from examples.a_aa import view
sep = '|'

keys  = list(view.keys())
skeys = list(map(lambda x: x.split(sep), keys))
lkeys = list(map(len, skeys))

root  = 'root'

depth = max(lkeys)
```

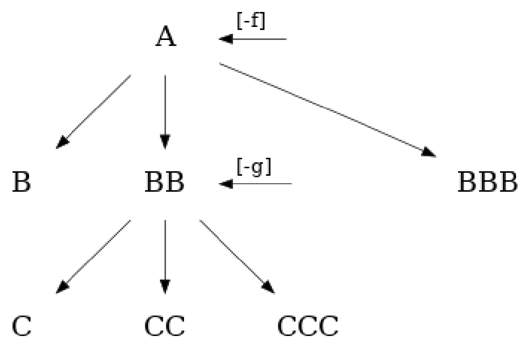
```python
while depth > 0:
    ix        = [i for i, e in enumerate(lkeys) if e == depth]
    k         = [key for i, key in enumerate(skeys) if i in ix]
    ancestors = list(map(lambda x: x[0:depth-1], k))
    parent    = list(map(lambda x: x[-1], ancestors))
    ancestors = list(map(lambda x: sep.join(x), ancestors))

    for a in set(ancestors):
        ii  = [i for i, e in enumerate(ancestors) if e == a]
        k   = [key for i, key in enumerate(skeys) if i in ix]
        # kk =  ### UNFINISHED - STOPPED HERE

    for key, ll, n in zip(keys, skeys, lkeys):
        if n == depth:
            ancestors = ll[:-1]
            parent    = ancestors[-1]
            ancestors = sep.join(ancestors)
    depth -= 1
```

```python
import numpy as np
import pandas as pd
from examples.a_aa import view

def to_tree_backwards(view, root_name='root', sep='|'):
    def make_key_frame(view, root_name):
        keys = list(view.keys())
        if not all(map(lambda key: key.startswith(root_name), keys)):
            keys = list(map(lambda key: root_name + sep + key, keys))
        d = pd.DataFrame({
            'orig_key': list(view.keys()),
            'key':      keys
        })
        d['skey']      = d['key'].apply(lambda x: x.split(sep))
        d['depth']     = d['skey'].apply(len)
        d['leaf']      = np.NaN
        d['parent']    = np.NaN
        d['ancestors'] = np.NaN
        return d

    d = make_key_frame(view, root_name)

    which = lambda ix: [i for i, e in enumerate(ix) if e]
    col   = lambda cn: which(d.columns==cn)[0]

    depth = max(d['depth'])
    while depth > 1:
        ix = d['depth']==depth
        d.loc[ix, 'leaf'] = d.loc[ix, 'skey'] \
            .apply(lambda x: x[-1])
        d.loc[ix, 'ancestors'] = d.loc[ix, 'skey'] \
            .apply(lambda x: sep.join(x[:-1]))
        d.loc[ix, 'parent'] = d.loc[ix, 'ancestors'] \
            .apply(lambda x: x.split(sep)[-1])
        for a in set(d['ancestors'].dropna()):
            ii = d['ancestors']==a
            parent = set(d.loc[ii, 'parent'])
            assert len(parent)==1
            parent = parent.pop()
            for i in which(ii):
                orig_key = d.iloc[i, col('orig_key')]
                key      = d.iloc[i, col('key')]
                leaf     = d.iloc[i, col('leaf')]
                children = d['leaf'].loc[d['ancestors']==key].to_list()
                d.iloc[i, col('leaf')] = Tree(
                    Tree.Value(
                        name=leaf if isinstance(leaf, str) else leaf.value.name,
                        key=orig_key, d=view
                    ),
                    children=children
                )
        depth -= 1

    return Tree(
        root_name,
```

```
        children=d.iloc[which(d['ancestors']==root_name), col('leaf')].to_list()
    )
```

4. Worked Examples

4.1. **The A-tree.** The “A tree” has three levels. As each node of the tree must necessarily occupy a positional argument of the command line, `[-f]` and `[-g]` are correspondingly *optional arguments* that attach to the nodes (`A` and `BB`, respectively, in the below diagram).



Let’s look at the Python code for `a.py`.

Consider the `dict` that represents the View component:

```
from examples.a import view

show(view)
```

```
'A':
    'callback':
    <function make_callback.<locals>.func at 0x7f166ae7a3a0>
    '-c':
    {'help': 'A [optional pi]', 'action': 'store_const', 'const': 3.141593}
    '-v|--verbose':
    {'help': 'A verbosity', 'action': 'store_true'}
'A|B':
    'callback':
    <function make_callback.<locals>.func at 0x7f166ad430d0>
    '-c':
    {'help': 'A B [optional pi]', 'action': 'store_const', 'const': 3.141593}
    '-v|--verbose':
    {'help': 'A B verbosity', 'action': 'store_true'}
'A|BB':
    'callback':
    <function make_callback.<locals>.func at 0x7f166ad43160>
    '-c':
    {'help': 'A BB [optional pi]', 'action': 'store_const', 'const': 3.141593}
    '-v|--verbose':
    {'help': 'A BB erbosity', 'action': 'store_true'}
'A|BB|C':
    'callback':
    <function make_callback.<locals>.func at 0x7f166ad431f0>
    '-c':
    {'help': 'A BB C [optional pi]', 'action': 'store_const', 'const': 3.141593}
    '-v|--verbose':
    {'help': 'A BB C verbosity', 'action': 'store_true'}
'A|BB|CC':
    'callback':
    <function make_callback.<locals>.func at 0x7f166ad43280>
    '-c':
    {'help': 'A BB CC [optional pi]', 'action': 'store_const', 'const': 3.141593}
    '-v|--verbose':
    {'help': 'A BB CC verbosity', 'action': 'store_true'}
'A|BB|CCC':
    'callback':
    <function make_callback.<locals>.func at 0x7f166ad43310>
    '-c':
    {'help': 'A BB CCC [optional pi]', 'action': 'store_const', 'const': 3.141593}
```

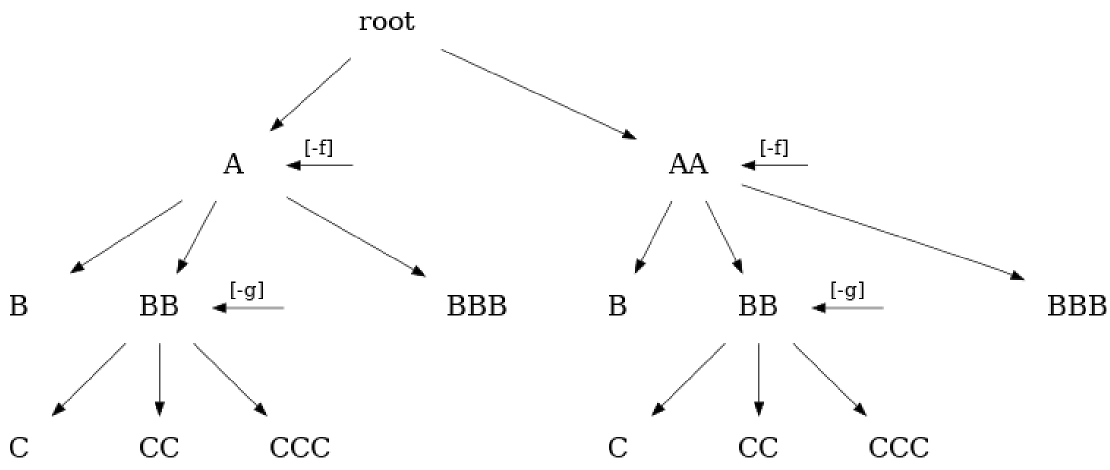
```
    '-v|--verbose':
    {'help': 'A BB CCC verbosity', 'action': 'store_true'}
'A|BBB':
    'callback':
    <function make_callback.<locals>.func at 0x7f166ad433a0>
    '-c':
    {'help': 'A BBB [optional pi]', 'action': 'store_const', 'const': 3.141593}
    '-v|--verbose':
    {'help': 'A BBB verbosity', 'action': 'store_true'}
```

and then the tree that the parsed `dict` is represented by:

```python
print(
    ParseArg(d=view, root_name='TODO').tree.show(quiet=True)
)
```

```
TODO
    A
        BB
            CCC
            C
            CC
        BBB
        B
```

4.2. **The A-AA tree.** The A-AA Tree simply extends the A Tree one level:



Let's look at the Python code for `a_aa.py`.

Consider the `dict` that represents the View component:

```python
from examples.a_aa import view

show(view)
```

```
'A':
    'callback':
    <function make_callback.<locals>.func at 0x7f166ad59310>
    '-c':
    {'help': 'A [optional pi]', 'action': 'store_const', 'const': 3.141593}
    '-v|--verbose':
    {'help': 'A verbosity', 'action': 'store_true'}
'A|B':
    'callback':
    <function make_callback.<locals>.func at 0x7f166ad59280>
    '-c':
    {'help': 'A B [optional pi]', 'action': 'store_const', 'const': 3.141593}
    '-v|--verbose':
    {'help': 'A B verbosity', 'action': 'store_true'}
'A|BB':
    'callback':
    <function make_callback.<locals>.func at 0x7f166ad59160>
    '-c':
    {'help': 'A BB [optional pi]', 'action': 'store_const', 'const': 3.141593}
    '-v|--verbose':
    {'help': 'A BB erbosity', 'action': 'store_true'}
'A|BB|C':
    'callback':
    <function make_callback.<locals>.func at 0x7f166ad59b80>
    '-c':
    {'help': 'A BB C [optional pi]', 'action': 'store_const', 'const': 3.141593}
    '-v|--verbose':
    {'help': 'A BB C verbosity', 'action': 'store_true'}
'A|BB|CC':
    'callback':
    <function make_callback.<locals>.func at 0x7f166ad59c10>
    '-c':
    {'help': 'A BB CC [optional pi]', 'action': 'store_const', 'const': 3.141593}
    '-v|--verbose':
    {'help': 'A BB CC verbosity', 'action': 'store_true'}
'A|BB|CCC':
    'callback':
    <function make_callback.<locals>.func at 0x7f166ad59ca0>
    '-c':
```

```
    {'help': 'A BB CCC [optional pi]', 'action': 'store_const', 'const': 3.141593}
    '-v|--verbose':
    {'help': 'A BB CCC verbosity', 'action': 'store_true'}
'A|BBB':
    'callback':
    <function make_callback.<locals>.func at 0x7f166ad59d30>
    '-c':
    {'help': 'A BBB [optional pi]', 'action': 'store_const', 'const': 3.141593}
    '-v|--verbose':
    {'help': 'A BBB verbosity', 'action': 'store_true'}
'AA':
    'callback':
    <function make_callback.<locals>.func at 0x7f166ad59dc0>
    '-c':
    {'help': 'AA [optional pi]', 'action': 'store_const', 'const': 3.141593}
    '-v|--verbose':
    {'help': 'AA verbosity', 'action': 'store_true'}
'AA|B':
    'callback':
    <function make_callback.<locals>.func at 0x7f166ad59e50>
    '-c':
    {'help': 'AA B [optional pi]', 'action': 'store_const', 'const': 3.141593}
    '-v|--verbose':
    {'help': 'AA B verbosity', 'action': 'store_true'}
'AA|BB':
    'callback':
    <function make_callback.<locals>.func at 0x7f166ad59ee0>
    '-c':
    {'help': 'AA BB [optional pi]', 'action': 'store_const', 'const': 3.141593}
    '-v|--verbose':
    {'help': 'AA BB verbosity', 'action': 'store_true'}
'AA|BB|C':
    'callback':
    <function make_callback.<locals>.func at 0x7f166ad59f70>
    '-c':
    {'help': 'AA BB C [optional pi]', 'action': 'store_const', 'const': 3.141593}
    '-v|--verbose':
    {'help': 'AA BB C verbosity', 'action': 'store_true'}
'AA|BB|CC':
    'callback':
    <function make_callback.<locals>.func at 0x7f166ace9040>
    '-c':
    {'help': 'AA BB CC [optional pi]', 'action': 'store_const', 'const': 3.141593}
    '-v|--verbose':
    {'help': 'AA BB CC verbosity', 'action': 'store_true'}
'AA|BB|CCC':
    'callback':
    <function make_callback.<locals>.func at 0x7f166ace90d0>
    '-c':
    {'help': 'AA BB CCC [optional pi]', 'action': 'store_const', 'const': 3.141593}
    '-v|--verbose':
    {'help': 'AA BB CCC verbosity', 'action': 'store_true'}
'AA|BBB':
    'callback':
    <function make_callback.<locals>.func at 0x7f166ace9160>
    '-c':
    {'help': 'AA BBB [optional pi]', 'action': 'store_const', 'const': 3.141593}
    '-v|--verbose':
    {'help': 'AA BBB verbosity', 'action': 'store_true'}
```

and then the tree that the parsed `dict` is represented by:

```
print(
    ParseArg(d=view, root_name='TODO').tree.show(quiet=True)
)
```

```
TODO
    A
        BB
            CCC
            C
            CC
        BBB
        B
    AA
```

```
        BB
            CCC
            C
            CC
        BBB
        B
```

We can now run the A Tree and the A-AA Tree examples, respectively:

```python
from examples.a import main
main()
```

```
NODE :: 'A':
--------------
usage: ipython A [-h] [-c] [-v] {BB,BBB,B} ...

positional arguments:
  {BB,BBB,B}

optional arguments:
  -h, --help     show this help message and exit
  -c             A [optional pi]
  -v, --verbose  A verbosity


'A':
----
        args: {'c': None, 'verbose': False}
        <Mock name='mock.A' id='139734259391360'>
'A -v':
-------
        args: {'c': None, 'verbose': True}
        <Mock name='mock.A' id='139734259391360'>
'A -c':
-------
        args: {'c': 3.141593, 'verbose': False}
        <Mock name='mock.A' id='139734259391360'>
'A -v -c':
----------
        args: {'c': 3.141593, 'verbose': True}
        <Mock name='mock.A' id='139734259391360'>


NODE :: 'A B':
----------------
usage: ipython A B [-h] [-c] [-v]

optional arguments:
  -h, --help     show this help message and exit
  -c             A B [optional pi]
  -v, --verbose  A B verbosity


'A B':
------
        args: {'c': None, 'verbose': False}
        <Mock name='mock.A_B' id='139734259392320'>
'A B -v':
---------
        args: {'c': None, 'verbose': True}
        <Mock name='mock.A_B' id='139734259392320'>
'A B -c':
---------
        args: {'c': 3.141593, 'verbose': False}
        <Mock name='mock.A_B' id='139734259392320'>
'A B -v -c':
------------
        args: {'c': 3.141593, 'verbose': True}
        <Mock name='mock.A_B' id='139734259392320'>


NODE :: 'A BB':
-----------------
```

```
usage: ipython A BB [-h] [-c] [-v] {CCC,C,CC} ...

positional arguments:
  {CCC,C,CC}

optional arguments:
  -h, --help     show this help message and exit
  -c             A BB [optional pi]
  -v, --verbose  A BB erbosity


'A BB':
-------
        args: {'c': None, 'verbose': False}
        <Mock name='mock.A_BB' id='139734258643200'>
'A BB -v':
----------
        args: {'c': None, 'verbose': True}
        <Mock name='mock.A_BB' id='139734258643200'>
'A BB -c':
----------
        args: {'c': 3.141593, 'verbose': False}
        <Mock name='mock.A_BB' id='139734258643200'>
'A BB -v -c':
-------------
        args: {'c': 3.141593, 'verbose': True}
        <Mock name='mock.A_BB' id='139734258643200'>


NODE :: 'A BB C':
-------------------
usage: ipython A BB C [-h] [-c] [-v]

optional arguments:
  -h, --help     show this help message and exit
  -c             A BB C [optional pi]
  -v, --verbose  A BB C verbosity


'A BB C':
---------
        args: {'c': None, 'verbose': False}
        <Mock name='mock.A_BB_C' id='139734258188544'>
'A BB C -v':
------------
        args: {'c': None, 'verbose': True}
        <Mock name='mock.A_BB_C' id='139734258188544'>
'A BB C -c':
------------
        args: {'c': 3.141593, 'verbose': False}
        <Mock name='mock.A_BB_C' id='139734258188544'>
'A BB C -v -c':
---------------
        args: {'c': 3.141593, 'verbose': True}
        <Mock name='mock.A_BB_C' id='139734258188544'>


NODE :: 'A BB CC':
--------------------
usage: ipython A BB CC [-h] [-c] [-v]

optional arguments:
  -h, --help     show this help message and exit
  -c             A BB CC [optional pi]
  -v, --verbose  A BB CC verbosity


'A BB CC':
----------
        args: {'c': None, 'verbose': False}
        <Mock name='mock.A_BB_CC' id='139734258189696'>
'A BB CC -v':
-------------
        args: {'c': None, 'verbose': True}
        <Mock name='mock.A_BB_CC' id='139734258189696'>
'A BB CC -c':
-------------
```

```
        args: {'c': 3.141593, 'verbose': False}
        <Mock name='mock.A_BB_CC' id='139734258189696'>
'A BB CC -v -c':
----------------
        args: {'c': 3.141593, 'verbose': True}
        <Mock name='mock.A_BB_CC' id='139734258189696'>


NODE :: 'A BB CCC':
---------------------
usage: ipython A BB CCC [-h] [-c] [-v]

optional arguments:
  -h, --help     show this help message and exit
  -c             A BB CCC [optional pi]
  -v, --verbose  A BB CCC verbosity


'A BB CCC':
-----------
        args: {'c': None, 'verbose': False}
        <Mock name='mock.A_BB_CCC' id='139734258190992'>
'A BB CCC -v':
--------------
        args: {'c': None, 'verbose': True}
        <Mock name='mock.A_BB_CCC' id='139734258190992'>
'A BB CCC -c':
--------------
        args: {'c': 3.141593, 'verbose': False}
        <Mock name='mock.A_BB_CCC' id='139734258190992'>
'A BB CCC -v -c':
-----------------
        args: {'c': 3.141593, 'verbose': True}
        <Mock name='mock.A_BB_CCC' id='139734258190992'>
```

```python
from examples.a_aa import main
main()
```

```


NODE :: 'A':
--------------
usage: ipython A [-h] [-c] [-v] {BB,BBB,B} ...

positional arguments:
  {BB,BBB,B}

optional arguments:
  -h, --help     show this help message and exit
  -c             A [optional pi]
  -v, --verbose  A verbosity


'A':
----
        args: {'c': None, 'verbose': False}
        <Mock name='mock.A' id='139734258340288'>
'A -v':
-------
        args: {'c': None, 'verbose': True}
        <Mock name='mock.A' id='139734258340288'>
'A -c':
-------
        args: {'c': 3.141593, 'verbose': False}
        <Mock name='mock.A' id='139734258340288'>
'A -v -c':
----------
        args: {'c': 3.141593, 'verbose': True}
        <Mock name='mock.A' id='139734258340288'>


NODE :: 'A B':
----------------
usage: ipython A B [-h] [-c] [-v]

optional arguments:
```

```
  -h, --help     show this help message and exit
  -c             A B [optional pi]
  -v, --verbose  A B verbosity


'A B':
------
        args: {'c': None, 'verbose': False}
        <Mock name='mock.A_B' id='139734258341680'>
'A B -v':
---------
        args: {'c': None, 'verbose': True}
        <Mock name='mock.A_B' id='139734258341680'>
'A B -c':
---------
        args: {'c': 3.141593, 'verbose': False}
        <Mock name='mock.A_B' id='139734258341680'>
'A B -v -c':
------------
        args: {'c': 3.141593, 'verbose': True}
        <Mock name='mock.A_B' id='139734258341680'>


NODE :: 'A BB':
-----------------
usage: ipython A BB [-h] [-c] [-v] {CCC,C,CC} ...

positional arguments:
  {CCC,C,CC}

optional arguments:
  -h, --help     show this help message and exit
  -c             A BB [optional pi]
  -v, --verbose  A BB erbosity


'A BB':
-------
        args: {'c': None, 'verbose': False}
        <Mock name='mock.A_BB' id='139734258383360'>
'A BB -v':
----------
        args: {'c': None, 'verbose': True}
        <Mock name='mock.A_BB' id='139734258383360'>
'A BB -c':
----------
        args: {'c': 3.141593, 'verbose': False}
        <Mock name='mock.A_BB' id='139734258383360'>
'A BB -v -c':
-------------
        args: {'c': 3.141593, 'verbose': True}
        <Mock name='mock.A_BB' id='139734258383360'>


NODE :: 'A BB C':
-------------------
usage: ipython A BB C [-h] [-c] [-v]

optional arguments:
  -h, --help     show this help message and exit
  -c             A BB C [optional pi]
  -v, --verbose  A BB C verbosity


'A BB C':
---------
        args: {'c': None, 'verbose': False}
        <Mock name='mock.A_BB_C' id='139734258382640'>
'A BB C -v':
------------
        args: {'c': None, 'verbose': True}
        <Mock name='mock.A_BB_C' id='139734258382640'>
'A BB C -c':
------------
        args: {'c': 3.141593, 'verbose': False}
        <Mock name='mock.A_BB_C' id='139734258382640'>
'A BB C -v -c':
```

```
---------------
        args: {'c': 3.141593, 'verbose': True}
        <Mock name='mock.A_BB_C' id='139734258382640'>


NODE :: 'A BB CC':
--------------------
usage: ipython A BB CC [-h] [-c] [-v]

optional arguments:
  -h, --help     show this help message and exit
  -c             A BB CC [optional pi]
  -v, --verbose  A BB CC verbosity


'A BB CC':
----------
        args: {'c': None, 'verbose': False}
        <Mock name='mock.A_BB_CC' id='139734258383552'>
'A BB CC -v':
-------------
        args: {'c': None, 'verbose': True}
        <Mock name='mock.A_BB_CC' id='139734258383552'>
'A BB CC -c':
-------------
        args: {'c': 3.141593, 'verbose': False}
        <Mock name='mock.A_BB_CC' id='139734258383552'>
'A BB CC -v -c':
----------------
        args: {'c': 3.141593, 'verbose': True}
        <Mock name='mock.A_BB_CC' id='139734258383552'>


NODE :: 'A BB CCC':
---------------------
usage: ipython A BB CCC [-h] [-c] [-v]

optional arguments:
  -h, --help     show this help message and exit
  -c             A BB CCC [optional pi]
  -v, --verbose  A BB CCC verbosity


'A BB CCC':
-----------
        args: {'c': None, 'verbose': False}
        <Mock name='mock.A_BB_CCC' id='139734258383504'>
'A BB CCC -v':
--------------
        args: {'c': None, 'verbose': True}
        <Mock name='mock.A_BB_CCC' id='139734258383504'>
'A BB CCC -c':
--------------
        args: {'c': 3.141593, 'verbose': False}
        <Mock name='mock.A_BB_CCC' id='139734258383504'>
'A BB CCC -v -c':
-----------------
        args: {'c': 3.141593, 'verbose': True}
        <Mock name='mock.A_BB_CCC' id='139734258383504'>


NODE :: 'AA':
---------------
usage: ipython AA [-h] [-c] [-v] {BB,BBB,B} ...

positional arguments:
  {BB,BBB,B}

optional arguments:
  -h, --help     show this help message and exit
  -c             AA [optional pi]
  -v, --verbose  AA verbosity


'AA':
-----
        args: {'c': None, 'verbose': False}
```

```
        <Mock name='mock.AA' id='139734258387408'>
'AA -v':
--------
        args: {'c': None, 'verbose': True}
        <Mock name='mock.AA' id='139734258387408'>
'AA -c':
--------
        args: {'c': 3.141593, 'verbose': False}
        <Mock name='mock.AA' id='139734258387408'>
'AA -v -c':
-----------
        args: {'c': 3.141593, 'verbose': True}
        <Mock name='mock.AA' id='139734258387408'>


NODE :: 'AA B':
-----------------
usage: ipython AA B [-h] [-c] [-v]

optional arguments:
  -h, --help     show this help message and exit
  -c             AA B [optional pi]
  -v, --verbose  AA B verbosity


'AA B':
-------
        args: {'c': None, 'verbose': False}
        <Mock name='mock.AA_B' id='139734258387792'>
'AA B -v':
----------
        args: {'c': None, 'verbose': True}
        <Mock name='mock.AA_B' id='139734258387792'>
'AA B -c':
----------
        args: {'c': 3.141593, 'verbose': False}
        <Mock name='mock.AA_B' id='139734258387792'>
'AA B -v -c':
-------------
        args: {'c': 3.141593, 'verbose': True}
        <Mock name='mock.AA_B' id='139734258387792'>


NODE :: 'AA BB':
------------------
usage: ipython AA BB [-h] [-c] [-v] {CCC,C,CC} ...

positional arguments:
  {CCC,C,CC}

optional arguments:
  -h, --help     show this help message and exit
  -c             AA BB [optional pi]
  -v, --verbose  AA BB verbosity


'AA BB':
--------
        args: {'c': None, 'verbose': False}
        <Mock name='mock.AA_BB' id='139734258386304'>
'AA BB -v':
-----------
        args: {'c': None, 'verbose': True}
        <Mock name='mock.AA_BB' id='139734258386304'>
'AA BB -c':
-----------
        args: {'c': 3.141593, 'verbose': False}
        <Mock name='mock.AA_BB' id='139734258386304'>
'AA BB -v -c':
--------------
        args: {'c': 3.141593, 'verbose': True}
        <Mock name='mock.AA_BB' id='139734258386304'>


NODE :: 'AA BB C':
--------------------
usage: ipython AA BB C [-h] [-c] [-v]
```

```
optional arguments:
  -h, --help     show this help message and exit
  -c             AA BB C [optional pi]
  -v, --verbose  AA BB C verbosity


'AA BB C':
----------
        args: {'c': None, 'verbose': False}
        <Mock name='mock.AA_BB_C' id='139734258387504'>
'AA BB C -v':
-------------
        args: {'c': None, 'verbose': True}
        <Mock name='mock.AA_BB_C' id='139734258387504'>
'AA BB C -c':
-------------
        args: {'c': 3.141593, 'verbose': False}
        <Mock name='mock.AA_BB_C' id='139734258387504'>
'AA BB C -v -c':
----------------
        args: {'c': 3.141593, 'verbose': True}
        <Mock name='mock.AA_BB_C' id='139734258387504'>


NODE :: 'AA BB CC':
---------------------
usage: ipython AA BB CC [-h] [-c] [-v]

optional arguments:
  -h, --help     show this help message and exit
  -c             AA BB CC [optional pi]
  -v, --verbose  AA BB CC verbosity


'AA BB CC':
-----------
        args: {'c': None, 'verbose': False}
        <Mock name='mock.AA_BB_CC' id='139734258387072'>
'AA BB CC -v':
--------------
        args: {'c': None, 'verbose': True}
        <Mock name='mock.AA_BB_CC' id='139734258387072'>
'AA BB CC -c':
--------------
        args: {'c': 3.141593, 'verbose': False}
        <Mock name='mock.AA_BB_CC' id='139734258387072'>
'AA BB CC -v -c':
-----------------
        args: {'c': 3.141593, 'verbose': True}
        <Mock name='mock.AA_BB_CC' id='139734258387072'>


NODE :: 'AA BB CCC':
----------------------
usage: ipython AA BB CCC [-h] [-c] [-v]

optional arguments:
  -h, --help     show this help message and exit
  -c             AA BB CCC [optional pi]
  -v, --verbose  AA BB CCC verbosity


'AA BB CCC':
------------
        args: {'c': None, 'verbose': False}
        <Mock name='mock.AA_BB_CCC' id='139734258387984'>
'AA BB CCC -v':
---------------
        args: {'c': None, 'verbose': True}
        <Mock name='mock.AA_BB_CCC' id='139734258387984'>
'AA BB CCC -c':
---------------
        args: {'c': 3.141593, 'verbose': False}
        <Mock name='mock.AA_BB_CCC' id='139734258387984'>
'AA BB CCC -v -c':
------------------
```

```
args: {'c': 3.141593, 'verbose': True}
<Mock name='mock.AA_BB_CCC' id='139734258387984'>
```

4.3. **Let's build the interface for a TO-DO app.** The `examples` folder in the source distribution contains a TO-DO app. The app

(1) illustrates a sufficiently realistic, but not overly complex, problem;

(2) illustrates operation of the MVC pattern in the wild;

(3) shows how `parsearg` neatly segments the View component of MVC with a `dict`.

Let's start with the outer layer of the onion. How do we interact with `todos.py`? First, we can create some users in a `User` table with the `create user` subcommand of `todos.py`. Note that we do not (yet) have a phone number for user `Dick`, nor do we have an email address for user `Harry`:

```
python todos.py create user Tom -e tom@email.com -p 212-555-1234
python todos.py create user Dick -e dick@email.com
python todos.py create user Harry -p 212-123-5555
```

```
'create user Tom -e tom@email.com -p 212-555-1234':
---------------------------------------------------
        SUCCESS
'create user Dick -e dick@email.com':
-------------------------------------
        SUCCESS
'create user Harry -p 212-123-5555':
------------------------------------
        SUCCESS
```

Second, create some TO-DOs in the `Todo` table with the `create todo` subcommand of `todos.py`.

```
python todos.py create todo Tom title1 -c description1 -d 2020-11-30
python todos.py create todo Tom title2 -c description2 --due-date=2020-12-31
python todos.py create todo Harry todo-1 --description=Christmas-party -d 2020-11-30
python todos.py create todo Harry todo-2 --description=New-Year-party
```

```
'create todo Tom title1 -c description1 -d 2020-11-30':
-------------------------------------------------------
        SUCCESS
'create todo Tom title2 -c description2 --due-date=2020-12-31':
---------------------------------------------------------------
        SUCCESS
'create todo Harry todo-1 --description=Christmas-party -d 2020-11-30':
-----------------------------------------------------------------------
        SUCCESS
'create todo Harry todo-2 --description=New-Year-party':
--------------------------------------------------------
        SUCCESS
```

Let's make some changes to the records entered so far. We can add an email address for user `Harry` (using the `update user email` subcommands) and a phone number for user `Dick` (using the `update user phone` subcommands):

```
python todos.py update user email Harry harry@email.com
python todos.py update user phone Dick 203-555-1212
```

```
'update user email Harry harry@email.com':
------------------------------------------
        SUCCESS
'update user phone Dick 203-555-1212':
--------------------------------------
        SUCCESS
```

Now update two of the TO-DOs, changing the `title` (using the `update todo title` subcommands) and the `description` (using the `update todo description` subcommands) in the fourth TO-DO:

```
python todos.py update todo title 4 most-important
python todos.py update todo description 4 2021-party
```

```
'update todo title 4 most-important':
-------------------------------------
        SUCCESS
'update todo description 4 2021-party':
---------------------------------------
        SUCCESS
```

```
python todos.py show users
python todos.py show todos
```

```
'show users':
-------------
        SUCCESS
'show todos':
-------------
        SUCCESS
```

The result of these commands is that the two tables (`User` and `Todo`) are populated in a SQLite database:

```
sqlite3 todo.db 'select * from User;'
```

```
Tom|tom@email.com|212-555-1234|2022-05-31 17:47:02
Dick|dick@email.com|203-555-1212|2022-05-31 17:47:02
Harry|harry@email.com|212-123-5555|2022-05-31 17:47:02
```

```
sqlite3 todo.db 'select * from Todo;'
```

```
33|title1|description1|2020-11-30|2022-05-31 17:47:02|Tom
34|title2|description2|2020-12-31|2022-05-31 17:47:02|Tom
35|todo-1|Christmas-party|2020-11-30|2022-05-31 17:47:02|Harry
36|todo-2|New-Year-party|None|2022-05-31 17:47:02|Harry
```

Let's look at the Python code for `todos.py`.

The View component is entirely contained within a single `dict`, *viz.* `view`, which has been formatted here for clarity using `parsearg.utils.show`:

```
from examples.todos import view
from parsearg.utils import show

show(view)
```

```
'purge|users':
    'callback':
    <function purge_users at 0x7fa92bce5430>
'purge|todos':
    'callback':
    <function purge_todos at 0x7fa92bcf0160>
'show|users':
    'callback':
    <function show_users at 0x7fa92bcf01f0>
'show|todos':
    'callback':
    <function show_todos at 0x7fa92bcf0280>
'create|user':
    'callback':
    <function create_user at 0x7fa92bcf0310>
    'name':
    {'help': 'create user name', 'action': 'store'}
    '-e|--email':
    {'help': "create user's email address", 'action': 'store', 'default': ''}
```

```
     '-p|--phone':
     {'help': "create user's phone number", 'action': 'store', 'default': ''}
 'create|todo':
     'callback':
     <function create_todo at 0x7fa92bcf03a0>
     'user':
     {'help': 'user name', 'action': 'store'}
     'title':
     {'help': 'title of to-do', 'action': 'store'}
     '-c|--description':
     {'help': 'description of to-do', 'action': 'store', 'default': ''}
     '-d|--due-date':
     {'help': 'due date for the to-do', 'action': 'store', 'default': None}
 'update|user|email':
     'callback':
     <function update_user_email at 0x7fa92bcf0430>
     'name':
     {'help': 'user name', 'action': 'store'}
     'email':
     {'help': 'user email', 'action': 'store'}
 'update|user|phone':
     'callback':
     <function update_user_phone at 0x7fa92bcf04c0>
     'name':
     {'help': 'user name', 'action': 'store'}
     'phone':
     {'help': 'user phone', 'action': 'store'}
 'update|todo|title':
     'callback':
     <function update_todo_title at 0x7fa92bcf0550>
     'id':
     {'help': 'ID of to-do', 'action': 'store'}
     'title':
     {'help': 'title of to-do', 'action': 'store'}
 'update|todo|description':
     'callback':
     <function update_todo_description at 0x7fa92bcf05e0>
     'id':
     {'help': 'ID of to-do', 'action': 'store'}
     'description':
     {'help': 'description of to-do', 'action': 'store'}
```

We can generate a tree view of the CLI design specified by the above `dict`, namely `view`, as follows:

```python
from parsearg import ParseArg

print(
    ParseArg(d=view, root_name='TODO').tree.show(quiet=True)
)
```

```
 TODO
     create
         user
         todo
     show
         todos
         users
     update
         user
             email
             phone
         todo
             description
             title
     purge
         todos
         users
```