- Implementation note -

- Bad variable name !! -

- In a number of places we have 'query', the

argument == list of TreeQL operations, but

we also have 'query' the instance variable,

referring to the parent object. This easily

confuses anyone trying to understand the

implementation.

=> Use a different name for the instance

variable referring to the parent object.

- Implementation note

We should add more comments to see which

methods are helpers, and which implement

the TreeQL operators. We should also reorder

the methods based on that.

The internal methods (like apply, mapself)

should have names with first char upper case.

Also better name for 'apply' is 'treeapply',

or 'treeop\_apply'. 'bool' is 'treefiler\_apply'

or something ...

- Should we save a node set before a query,

and restore if an error occurs ?

= query is like transaction =

= new node set happens only if successful =

- should we have push/pop operations to save/restore

node sets while the query executes ?

- this would essentially start to add

a stack manipulation language to the

TreeQL (push, pop, dup, swap, rot, ...)

- does it make sense to 'precompile' queries

into a tree ops, for faster execution

(saving, using precompiled queries ?)

- Tree ops -

tcllib tree has an 'ancestors' operation.

IMHO it should be used.

NOTE: When we reimpl. tree in snit we can use 'info methods' to

check for existence of 'ancestors', and fall back to the local

iteration via 'parent' if it is not present.

- list of left siblings can be had easier

(we have an 'index' op!)

- descendants can be had from tree.

(and|or|not)q use helpers an|or|not

given their definition they can be

regular operators as well.

> > \*\* Operator 'string' is not a mutator.

> > It is a string acessor. Does not change the tree, leaves

> > results in the node set => Definitely accessor.

>

> Quite right.

Ok, will move it in my docs.

operator reverse

> > \*\* Ideas, way out ...

> >

> > -- Keep not a node set, but a stack of sets, add

> > operations:

> > push, pop, dup, swap, rot

>

> Nice idea. It's already sort of implicitly there, though, in the

> andq/orq/notq stuff.

Right. I noticed that for SQ a subquery we have the equivalence

(andq SQ)

<=> (subquery SQ and)

if we also assume that there is some internal storage both 'subquery'

and 'and' know and use for the temp node set. Extending this storage

to a stack is conceptually easy and clean. Etc. for the other operators.

> OTOH, rather than duplicating stack functionality, a nice API to mix the

> struct::stack and other objects might be better. I've left the ability

> to set and query the nodeset, so this isn't foreclosed.

Oh, you mean a special object keeping the actual node and implementing

the access. Again defaults to self, but could be a specific rapper to

stack ? Hm.

(I had not thought of using the struct::stack, but its use is sort of

obvious. (I guess I tend to forget stack and queue objects, because

they can be implemented so lightweight with Tcl lists when and

whereever they are needed)).

> > I am not sure if adding this type of stack operators

> > will up the expressivity of the TreeQL very much. It

> > becomes a bit more forth-like.

>

> I'd suggest it would be good to provide a bunch of parent QL types for

> that.

Now that I truly understand the parent stuff, yes, this can be layered

on top of the basic query. And should be, so people can choose what

they need and how much baggage is required.

Yes, definitely use layering here to keep stuff simple and modular.

> > -- Precompile queries to Tcl code doing all ops directly

> > in proper sequence ?

> > (If we allow a q-object to save such queries, and

> > execute on demand. Maybe store in files ?)

>

> I \*really\* like that idea. It does become forth-like :)

The main loop of the query interpreter is already similar to a

inner-loop, inner-interp of a forth system. For this we have to tell

to not interpret the operator, but to run a compile method for the

operator, which adds Tcl code to a script, and this can be saved and

used later. ... Depending on the complexitiy of the created code the

script may depend on the infrastructure of a query object, or be

completely independent, i.e. require only access to a tree object. If

the latter then it becomes a 'hardwired query object', maybe a new

snit class ?

Note, this type of compiler can also be added through the 'parent

query' stuff. ... Heh, this could make things even more easy. Have the

user play with queries, then attach a parent query to your object, run

the queries again to get compiled, save ... The only not so modular

thing is that the compiler has to know how the operators are

ipmlemented, and this may not gel with an extended query having stack

operation. ... Or can we use the snit method introspection here as

well ? Get methods available, get method body, use that in the

generated code ... But then methods have to be written carefully so

that we can extract the actual operation from the code accessing node

set and such. ... Or we split the current methods into true

functionality and a interpreter shim. The compiler needs only the

functionality, and uses his own mgmt code ... Oh the ideas ...

Ok, stop now before completely flying off the deep end. I will put all

this into a file IDEAS.txt of the module and place it under revision

control, for future work.

\*\* Proposal: Add a `node set operator` -> "reverse".

set nodes [struct::list reverse $nodes]

This makes the description of a number of

operators, like "backward" easier.

backward <==> 'prev subtree reverse'

\*\* Ideas, way out ...

-- Keep not a node set, but a stack of sets, add

operations:

push, pop, dup, swap, rot

I am not sure if adding this type of stack operators

will up the expressivity of the TreeQL very much. It

becomes a bit more forth-like.

-- Precompile queries to Tcl code doing all ops directly

in proper sequence ?

(If we allow a q-object to save such queries, and

execute on demand. Maybe store in files ?)