http://www.vpri.org/pdf/tr2011001_final_worlds.pdf

Isolation

Objects in a world have no implicit dependencies on objects in other worlds.

PEG Makes This Easy

The ideas in this paper seep into many parts of a GPL (General Purpose programming Language).

PEG - OMeta (ancestor of Ohm-JS) - makes it easy to rewrite the syntax of any GPL, hence, making it easy to enact these changes.

Generates Ideas

I think that I originally saw this in the Ohm-JS thesis.

This shows what is possible when you elide details and "free your mind" to think about higher-level concepts.

Worlds is not directly related to Ohm-JS, but springs forth as an idea of how to use Ohm-JS to build new kinds of solutions to problems.

"… An important class of problems have to perform *speculations* and *experiments*, often in parallel, to discover how to proceed. …" [*pt: I call this Design, Brainstorming, Architecture, etc.)*]

paraphrase: Try/Catch is a subset of undo. [*pt: Kind of like Greenspun's 10th Rule of undo.*]

Web surfing … back button … exploration …

"… This is somewhat similar to *transactions* …"

"… Worlds are first-class structures …"

[*pt: I believe that every useful programming concept needs to be made explicit, e.g. GC (garbage collection), OO, etc. Making something first-class in a language is but one way to make concepts explicit. (GC is not usually first-class, OO objects are usually first-class)*]

Worlds realized in

- JavaScript
- Squeak (Smalltalk).

- *Sprout* a world (instantiate from prototype),
- make changes
- *commit* changes back to parent (if possible, see below for algorithm)
- field access
    - lookups
        - lookup in local scope
        - chain upwards through parents, if not found in local scope.
    - updates
        - always done locally
        - *commit* operation pushes changes to parent.
    - [*pt: is this similar to pre-CL Lisps and/or special variables in CL? "dynamic scoping"*]

```
A = thisWorld;
p = new Point (1, 2);

B = A.sprout ();
in B {
  p.y = 3
}

C = A.sprout ();
in C {
  p.y = 7;
}

C.commit ();
```

`P.y` is 2 in A, while `p.y` is 3 in B, while `p.y` is 7 in C.

`P.y in A` becomes 7 after commit.

- No Surprises
- Consistency

- scoped methods

- turnstile notation

- WObject
- WWorld

WObject

Each slot of each object contains 2 fields:

- Reads
- Writes

[*pt: We are accustomed to thinking of variables as containing exactly 1 field, but WObject*s *contain 2 fields*]

Each slot is characterized by

- don't know - '?'
- a value

WWorld

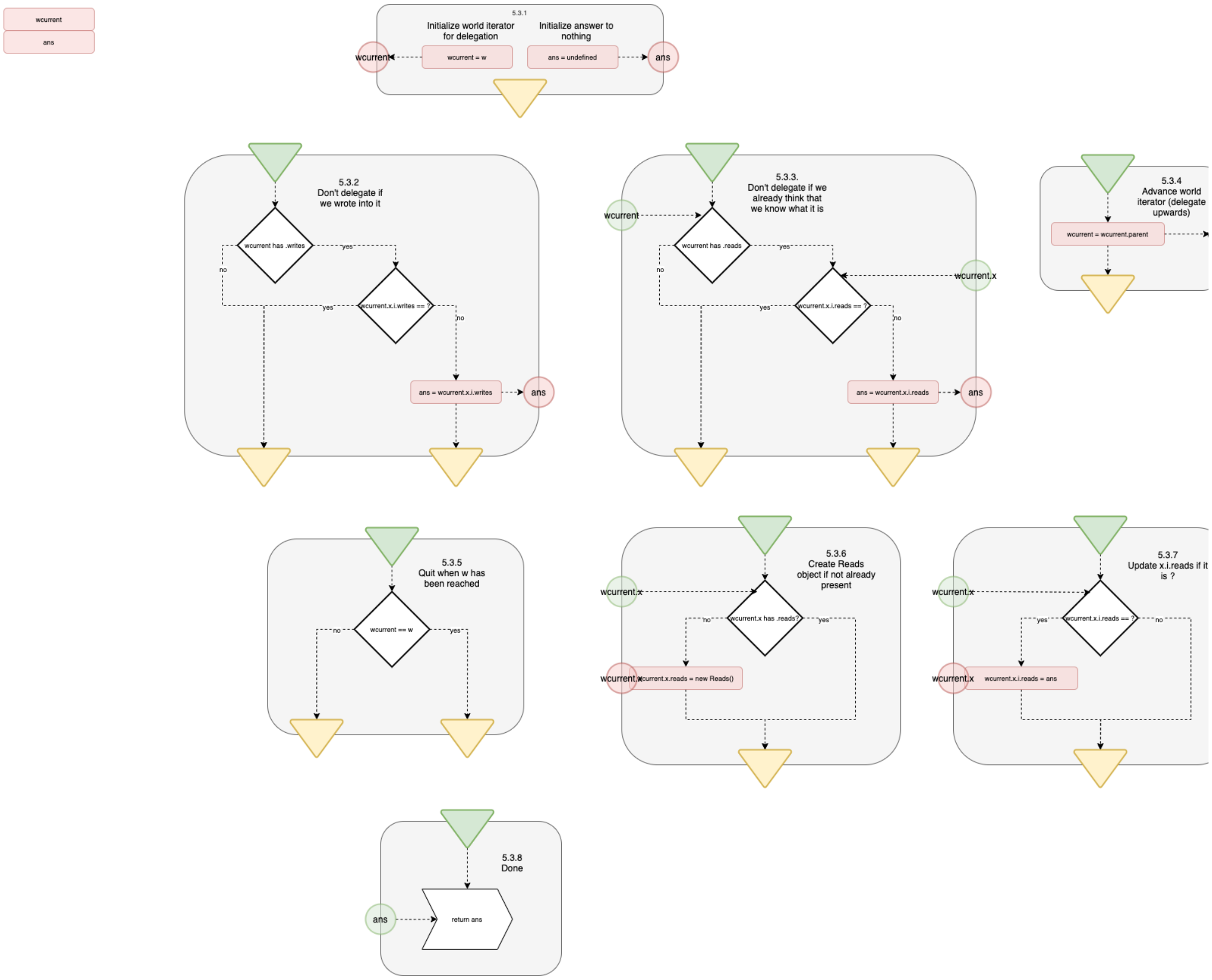
collection of WObjects

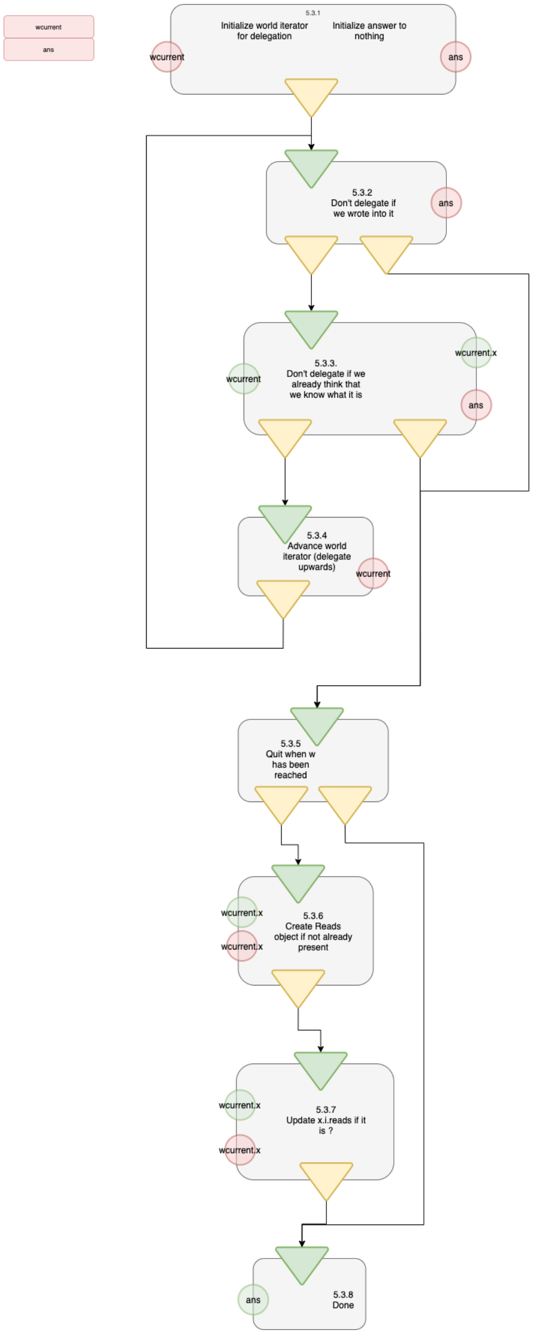
5.2 Update w.x.i

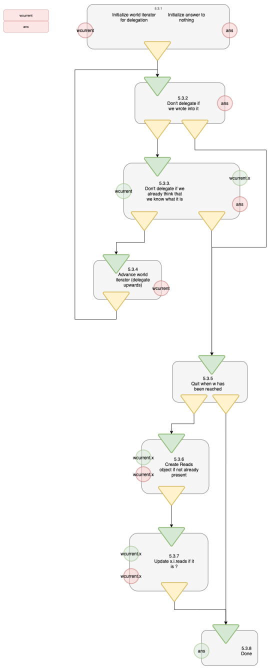
5.2.1 (optimization) - create *writes* for *x*

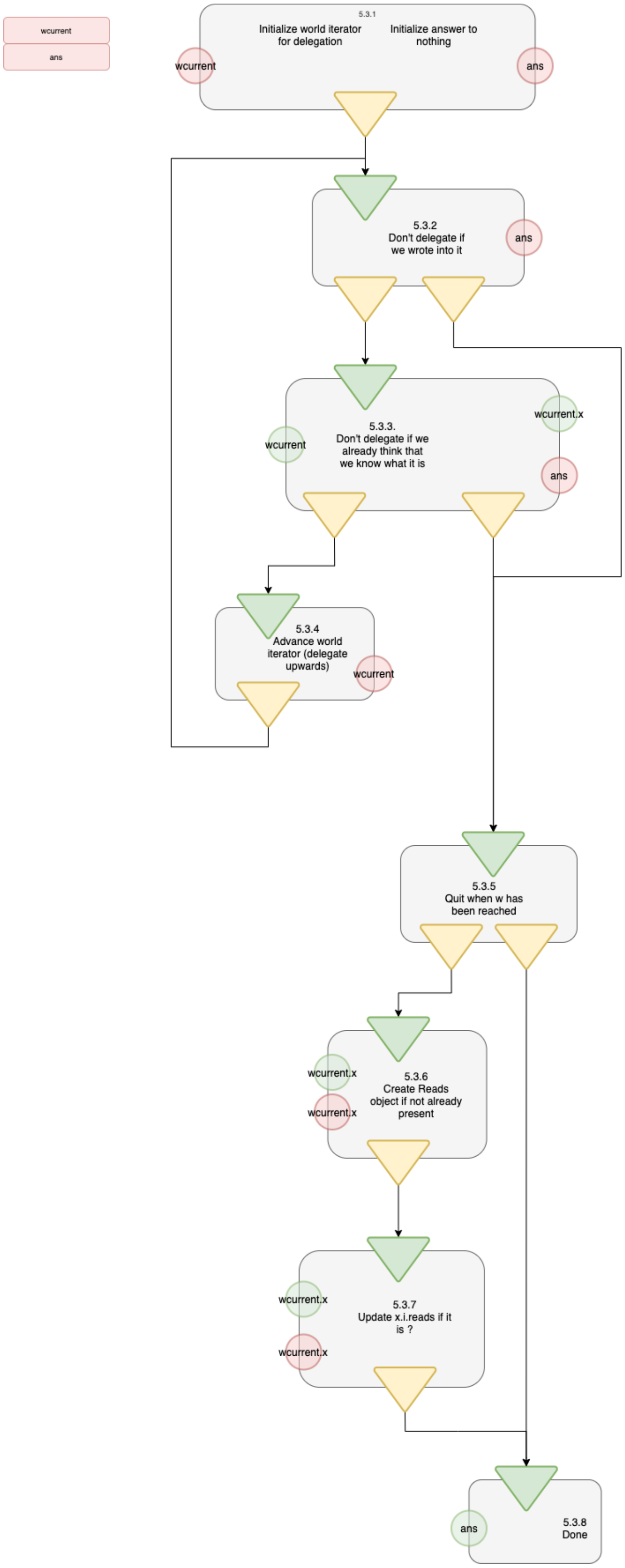
5.2.2 write *v* into *w.x.i.writes*

8-step algorithm, much like a flowchart




wcurrent
ans
5.3.1
Initialize world iterator for delegation
Initialize answer to nothing
wcurrent
ans
5.3.2
Don't delegate if we wrote into it
ans
5.3.3.
Don't delegate if we already think that we know what it is
wcurrent.x
wcurrent
ans
5.3.4
Advance world iterator (delegate upwards)
wcurrent
5.3.5
Quit when w has been reached
wcurrent.x
wcurrent.x
5.3.6
Create Reads object if not already present
wcurrent.x
5.3.7
Update x.i.reads if it is ?
wcurrent.x
ans
5.3.8
Done


wcurrent
ans
5.3.1
Initialize world iterator for delegation
Initialize answer to nothing
wcurrent
ans
5.3.2
Don't delegate if we wrote into it
ans
5.3.3.
Don't delegate if we already think that we know what it is
wcurrent
wcurrent.x
ans
5.3.4
Advance world iterator (delegate upwards)
wcurrent
5.3.5
Quit when w has been reached
5.3.6
Create Reads object if not already present
wcurrent.x
wcurrent.x
5.3.7
Update x.i.reads if it is ?
wcurrent.x
wcurrent.x
5.3.8
Done
ans


wcurrent
ans
5.3.1
Initialize world iterator for delegation
Initialize answer to nothing
wcurrent
ans
5.3.2
Don't delegate if we wrote into it
ans
5.3.3.
Don't delegate if we already think that we know what it is
wcurrent
wcurrent.x
ans
5.3.4
Advance world iterator (delegate upwards)
wcurrent
5.3.5
Quit when w has been reached
5.3.6
Create Reads object if not already present
wcurrent.x
wcurrent.x
5.3.7
Update x.i.reads if it is ?
wcurrent.x
wcurrent.x
ans
5.3.8
Done

Commit 5.5

5.5.1:

```
either { child.reads.field == ?}
or { child.reads.field == parent.field }
```

5.5.2:

```
forall field { parent.writes.field := child.writes.field }
```

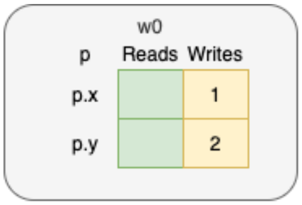
5.5.3:

```
forall field { parent.reads.field := child.reads.field }
```

** I don't yet understand the exception - is it necessary or merely an optimization?*

5.5.4:

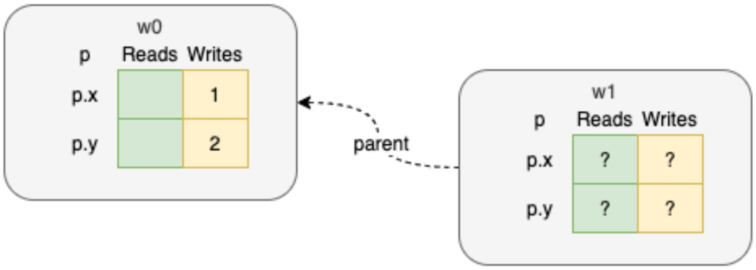
```
clear child
```



w0

p	Reads	Writes
p.x		1
p.y		2

w1 := w0.sprout()



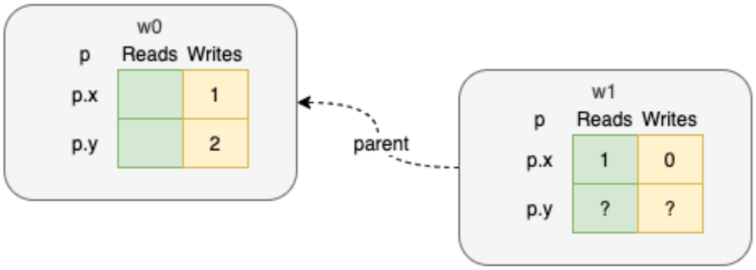
w0

p	Reads	Writes
p.x		1
p.y		2

w1

p	Reads	Writes
p.x	?	?
p.y	?	?

in w2 { p.x := p.x - 1 }



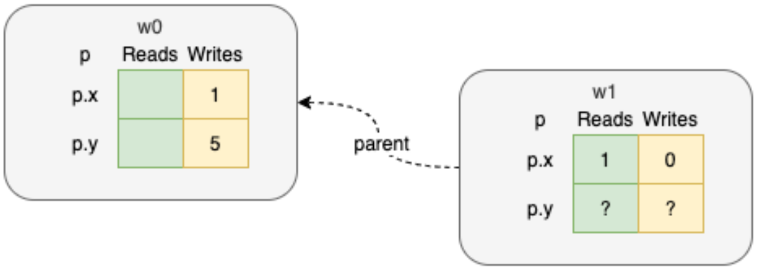
w0

p	Reads	Writes
p.x		1
p.y		2

w1

p	Reads	Writes
p.x	1	0
p.y	?	?

in w0 { p.y := 5 }



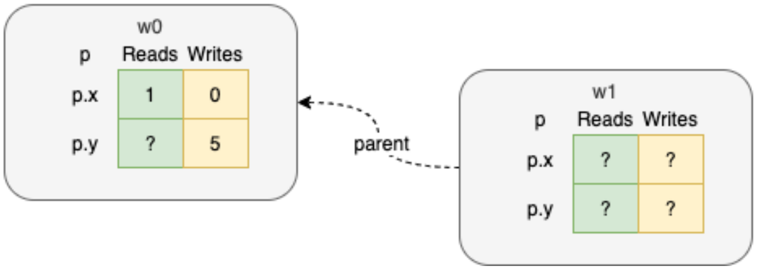
w0

p	Reads	Writes
p.x		1
p.y		5

w1

p	Reads	Writes
p.x	1	0
p.y	?	?

w1.commit

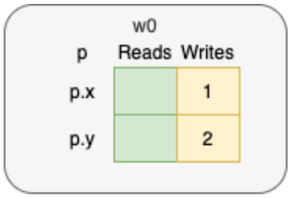


w0

p	Reads	Writes
p.x	1	0
p.y	?	5

w1

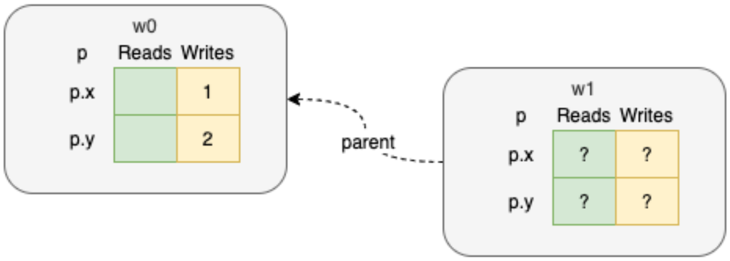
p	Reads	Writes
p.x	?	?
p.y	?	?



w0

p	Reads	Writes
p.x		1
p.y		2

w1 := w0.sprout()



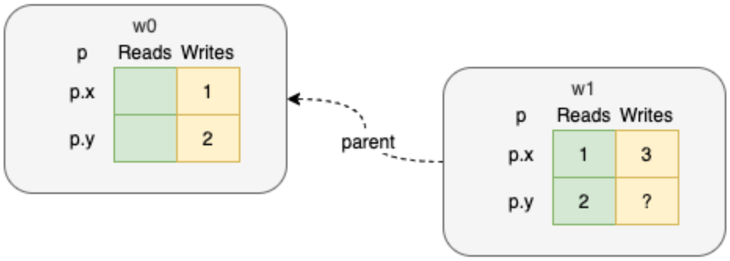
w0

p	Reads	Writes
p.x		1
p.y		2

w1

p	Reads	Writes
p.x	?	?
p.y	?	?

in w2 { p.x := p.x + p.y }



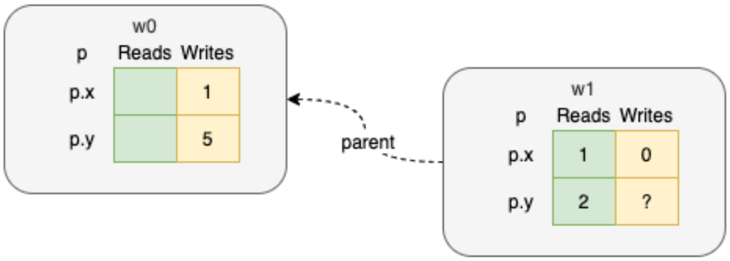
w0

p	Reads	Writes
p.x		1
p.y		2

w1

p	Reads	Writes
p.x	1	3
p.y	2	?

in w0 { p.y := 5 }



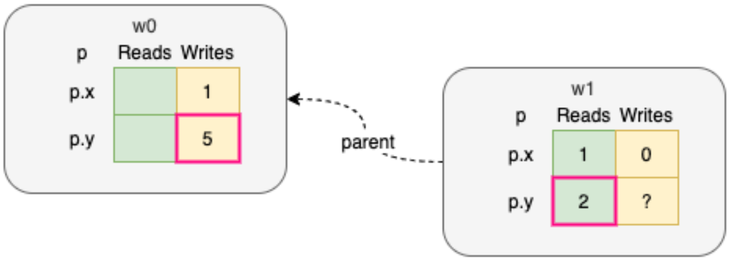
w0

p	Reads	Writes
p.x		1
p.y		5

w1

p	Reads	Writes
p.x	1	0
p.y	2	?

w1.commit



w0

p	Reads	Writes
p.x		1
p.y		5

w1

p	Reads	Writes
p.x	1	0
p.y	2	?

Commit fails 5.5.1

Fig. 6 shows 3 sub-worlds, where "1" is drawn in sub-world-1, "2" is drawn in sub-world-2 and "3" is drawn in sub-world-3.

In sub-world 3, the serifs are removed from the "1". This results in a few changes to the bitmap, (seen as white squares) but leaves much of the bitmap pixels in a "don't know" state.

The sub-world-3 bitmap is "flattened" to produce a Squeak-compatible bitmap and to improve editor performance.

OMeta is a PEG parser technology (which inspired Ohm-JS).

PEG uses backtracking to parse incoming grammars, e.g. (A | B) tries to parse an A rule, if that fails, it backtracks and tries to parse a B rule.

Section 7 describes the experiment of replacing backtracking in OMeta with Worlds.

The conclusion:

> And thus, with very little additional complexity, worlds can be use to make the use of side effects safer and easier to reason about in the presence of backtracking.

Touches on:

- STM

- revisions and isolation types
- GL programming language (snapshots of the store)
- Contextual values
- Us, COP (Context-Oriented Programming)
- FDS (Functional Data Structures)
- concurrency control

- Worlds/JS, Worlds/Squeak
- hardware-assist for Worlds?
- infinite undo?
- persistence?
- in-memory vs. network side effects?

Contains an Introductory graphic that might help motivation.

http://www.vpri.org/pdf/rn2008001_worlds.pdf

Meet Expectations?

The Worlds concept partially meets my expectations and shows future promise.

Has the potential to subsume GC (Garbage Collection).

Scoped GC (ignoring commit).

Akin to UNIX® processes (which "clean up" when apps die). I like Worlds and UNIX® processes (for *isolation*), but, also, see potential for further improvement.

I see this as a basic technology that can be shaped (further scoped) to provide multiple notations (languages).

Commit of many variables has the potential to break locality-of-reference.

Is the implementation easier/quicker if *commit* is dropped?

Is *commit* needed?

Javascript nearly meets the needs of creating worlds, because all JS variables are contained in the *global* (node.js) object and the *window* (HTML) object.

For example

```
var x = 5;
```

is semantically equivalent to

```
var global.x = 5;
```

If we could change the value of `global`, we could create Worlds.

I *think* that the paper was based on the use of OMeta (PEG) to pre-process code and to transpile Worlds-based-code into stock JS.

Essence of idea might lead to *isolation*, and isolated components.

Restrict coupling only to *ports*, not all variables.

Might be easy to build in, say, JavaScript'. In JS, all variables belong to a context, all variables are fields of a JS *object* { name:value, … }

Thought: implement separate worlds but drop *commit.* Would this be enough to provide UNIX®-like *isolation*?

TXL (txl.ca) - functional language for parsing and rewriting syntax.