specifying GroovyMarkup

GroovyMarkup is an exciting feature of Groovy. It promises to ease the interface to symbolic tree structures, such as XML and widget hierarchies. This will enhance their scriptability!

Strengths of the design include the following:

- Simple, quasi-declarative specification of attributed tree structures.
- Synergy (close correspondence) between Groovy program syntax and tree-specification syntax.
- Natural intermingling of imperative scripting code with symbolic tree specification.
- Application-specific scoping of tree tags (using "catch-all" delegates).
- (Likely) application to a Groovy AST formalism for a Groovy syntax-extension facility.
- *(more?)*

The present design, however, suffers from several defects, most or all of which should be removed in the mature design.

- Radical ambiguity between Groovy names and syntax-tree names (see property versus field scoping).
- Inadequate spelling power for attribute and tag names (e.g., XML name alphabet >> Groovy name alphabet).
- Processing of tag names via "last-chance lookup", which makes unintentional interception likely.
- *(more?)*

The basic requirement for GroovyMarkup is that, for certain parts of a Groovy script, the name-lookup mechanism (the current scope) be "hijacked" by a foreign naming scheme. The native scheme is, of course, Groovy's normal program scoping, which is similar to that of Java or any other lexically scoped language.

A second requirement is that the programmer be able to introduce normal Groovy names, mingling them into a structural template expressed in a (quasi-)declarative syntax. This is highly groovy and useful, because it allows the programmer to make use of two languages at once. On a smaller scale, the shell-like string syntax `"hello, $name"` is nice for exactly the same reasons.

Other interesting examples of this "mix two languages" tactic are:

- Shell-type string-construction, as in `"hello, $name"` (mix parameter subst. into a string body template)
- Java Server Pages (mix Java into an HTML template)
- Lisp backquote-comma (mix Lisp into an S-expr template)

jrose Proposal

Fix the ambiguity problem by taking a cue from Lisp and other examples cited above: Require that markup code be introduced by a "quasiquote" operator which says "everything I quote names syntax-tree elements, by default". Introduce a corresponding "unquote" operator which says "despite the fact that I occur inside a quasiquote, names under me refer to Groovy program elements, by default."

Fix the spelling power problem by allowing tag and attribute names to be not only Groovy identifiers, but also quoted strings or even arbitrary string-valued expressions. This can be done unambiguously and simply by allowing such things uniformly after dot (`x.n`) and before binary colon (`n.y`).

Fix the problem of unintentional interception by simplifying the rules for identifiers under a quasiquote. Do not pass them through Groovy's generic MOP (metaobject protocol), but through a specific API designed for them alone.

The unquote operator should obviously be spelled "$", since it is already used as an unquote in Groovy strings. The quasiquote operator should play well with curly braces for ease of reading. Possibilities:

- JSP-like flowered brackets: `builder.{% person(name: n){ ...} %}`
- or maybe XML-like angle brackets: `builder.<person(name: n){ ...}>`

The specific API for quasiquoted names can be quite simple. I propose that it be called 'build', and that it take all the GroovyMarkup arguments specified, plus a prepended (non-keyworded) argument, which is the tag itself. Every method application `m.f(...args)` in the markup is transformed to a builder call `b.build(m,...args)`.

Example:

```
someBuilder.{% person(name:'Fred') { pet(name:'Dino') } %}
// equivalent to:
someBuilder.build('person', name:'Fred') { someBuilder.build('pet', name:'Dino') }>
```

Longer Example:

```
let someBuilder = new NodeBuilder()
let uidTag = 'uid:id', uid = 123, petTag = 'dog'

someBuilder.{%
people(kind='folks' groovy='true') {
  // first a person who is constant:
  person($uidTag:$uid, name:'James', cheese:'edam') {
    project(name:'groovy')
    project name:'geronimo')
    ($petTag)(name:'Fido')
  }
  // now, some other people, rounded up on the fly:
  ${
    for (n : findOtherNames() {pn|filterPotentialName(pn)}) {
      println "Making cheddar-loving person for $n..."
      someBuilder.{%
        person('uid:id':234, name:$n, cheese:'cheddar') {
          project(name:'groovy')
          'pet-project'('pet-name':'drools')
        }
      %}
    }
  }
%}
```

Missing bits from this proposal:

- It is likely the builder API cannot be this simple.
- Need a specification for the meaning of other quasiquoted expressions `x`, `x.y`, `x.y(a )`, `x=y`, `x+y`, etc.
- Tags cannot be always only strings; need to play with XML namespaces when needed.
- Need to explore synergy with Groovy syntax extension.
- Need to inquire whether direct inclusion of XML syntax is desirable, despite trends toward angle-bracket fatigue.