Authz Improvements

Improved authz

Goals

The current (1.9) implementation of authz is lacking in two areas:

1. Wildcard support and performance
    - Performance when a large number of paths needs to be checked. Affected operations are mainly checkout / export and log.
    - Support for wildcards. Subversion should support
        - "*" for single (exactly one), arbitrary path segments (with no "/" in them) and
        - "**" for arbitrary number (zero to infinite) of path segments.
        - Classic wildcard patterns like "*foo*.bar", including escapement via the "\" prefix shall be supported. The asterisks in there match zero to many characters other than "/" making the whole segment form "/*/" a mere special case of this.
          All wildcard usage applies to full path segments only, i.e. a '*' never matches a '/' except for the case of "/**/" where it matches zero to many full segments. For example, "/*/**/*" will match any path that contains at least 2 segments and is equivalent to "/**/*/*" as well as "/*/*/**".
2. Better access control
    - The right to know about the existence of a node (a.k.a. lookup access rights and/or directory traversal rights) is implied by read access and cannot be manipulated separately.
        - See Issue 3380 for previous discussion of this topic;
        - the authz-overhaul branch for an attempt at implementing this distinction.
        - This thread on the dev@ mailing list is a recent example of the problems caused by implicit lookup access.

Wildcard Support and Performance

Terminology

A **path** consists of **segments**, separated by "/". Whildcards are valid segments and their interpretation depends on the context.

An **access control list (ACL)** contains a number of **access control elements (ACE)**, each describing which **users** shall have what **access rights**. Users might be specified indirectly by using **groups**.

A **path rule** specifies what ACL applies to a given path.

Inheritance and Disambiguation

These are the rules as of today, with rule 7 added to support ambiguous path matches caused by pattern matching. Please note the difference between *matching* (does a path fully match a given pattern?) and *applying to* (governing the access rights).

1. An ACL is relevant to a user if the user, one of their aliases or groups that they are a member of is mentioned by at least one ACE in that ACL.
2. Only path rules with ACLs relevant to the given user may match a path.
3. If a path rule matches a given repository path, its ACL applies to that path.
4. If no path rule matches a given repository path, the parent path's ACL applies.
5. If no ACL is given for the repository root, a default ACL denying everybody access to the root path, applies.
6. If repository-specific path rules as well as global path rules match a given path, only the repository-specific ones will be considered.
7. If multiple path rules match a given repository path, only the one specified last in the authz file shall apply.
8. If multiple ACEs of a given ACL apply to a user, the union of all individually granted access rights is granted.

Design

The idea is to use combine the following approaches:

- Preprocessed, tree-like data structures applied to segmented paths
- Reduction of the tree to what applies to the current user
- Pre-calculate recursive rights for early exit

General Workflow

Putting caching aside, the workflow involved three data models, building on top of each other.

- Parsing an authz file (from file system or repository), validating its contents and creating a pre-processed in-memory representation. In comparison to the old `svn_config_t` based code, additional restrictions apply:
    - No rule may appear more than once in the authz file.
    - Value placeholders (`%(name)s`) are not expanded.
- Filtered path rule tree
    - prefix tree with one node per segment
    - created on demand per user and repository
    - contains only rules that apply to the respective user and repository
    - multiple instances of that being cached in the svn_authz_t structure alongside the single "full model"
    - ACLs being reduced to access rights + order ID

        - each node knows min / max rights on all sub-nodes
    - Lookup state
        - access rights accordingly the latest matching path rule
        - list of tree nodes that may match sub-paths as we may need to follow multiple patterns
        - temporary data structure thats reused between queries to save on allocation and construction overhead

Data models

These are persistent in the sense that we will cache and reuse them. They do not cover transient data models that various algorithms may use e.g. during authz parsing.

Unless indicated otherwise, all collections are ordered.

Only abstract types / the interface view is given here; the actual implementation may be different.

Filtered path rule tree

```
filtered-tree :=
    root      : filtered-node       // exists due to default path rule

filtered-node :=
    segment   : string              // empty for root
    access    : ref to access-type  // empty if no path ends here
    min-rights: none | r | rw       // user's minimum rights on any path in the sub-tree
    max-rights: none | r | rw       // user's maximum rights on any path in the sub-tree
    repeat    : boolean             // set on nodes for "**" segments

    sub-nodes : { map following segment => filtered-node }[0..*]
                                    // one node per following segment with no wildcards
                                    // empty if no such rules exist

    // the following will be aggregated into a sub-structure to
    // facilitate a quick "has it pattern sub-segments?" check
    any       : filtered-node       // empty if no path rule with "*" in following segment
    any-var   : filtered-node       // empty if no path rule with "**" in following segment
    prefixed  : { list of filtered-node }[0..*]
                                    // empty if no path rule like "bar*" in following segment
    suffixed  : { list of filtered-node }[0..*]
                                    // empty if no path rule like "*bar" in following segment
    complex   : { list of filtered-node }[0..*]
                                    // empty if no path rule with complex wildcard pattern
                                    // like "*for*bar" in following segment

access-type :=
    order-id  : integer             // increases with declaration order in the file
    rights    : none | r | rw
```

Lookup state

```
lookup-state :=
    access    : ref to access-type  // user's rights for this path
    min-rights: none | r | rw       // user's minimum rights on any path in the sub-tree
                                    // (aggregated over RIGHTS and all nodes in CURRENT)
    max-rights: none | r | rw       // user's maximum rights on any path in the sub-tree
                                    // (aggregated over RIGHTS and all nodes in CURRENT)
    current   : { ref to filtered-node }[0..*]
                                    // sub nodes of these may match the next segment
```

Algorithms

Normalization

Wildcard sequences in paths shall be normalized internally. This is merely done to reduce matching costs later on and some of the matching code may rely on normalized pattern.

```
    // Variable length wildcards shall be the last in a sequence
    while path contains "/**/*/"
        replace occurrence with "/*/**/"

    // Consecutive variable length wildcards are redundant
    while path contains "/**/**/"
        replace occurrence with "/**/"
```

Lookup

Since there is always an implicit path rule at the root for all users, lookup is always necessary.

```
    init(tree : ref to filtered-tree):
        return new lookup-state
            .rights = tree.root.rule
            .current = { tree.root }, if has_subnodes(tree.root)
                       { }, otherwise
```

Checking whether we need to continue in our path matching is trivial, just check whether there are any potential matches to sub-paths.

```
    done(state : ref to lookup-state):
        return is_empty(state.current)
```

One step in the lookup iteration, i.e. matching the next segment, is relatively simple as well. For simplicity, we ignore the min/max rights optimization here.

```
    latest(lhs : ref to access-type, rhs : ref to access-type):
        if is_empty(lhs)
            return rhs
        if is_empty(rhs) or lhs.order-id > rhs.order-id
            return lhs
        return rhs

    step(state : ref to lookup-state, segment : string):
        next = { }
        access = empty
        foreach node in state.current:
            if node.repeat:
                next += node
                access = latest(access, node.access)
            foreach sub-node in all-subnodes(node):  // simplified
                if matches(sub-node, segment):           // simplified
                    next += sub-node
                    access = latest(access, sub-node.access)

        if not is_empty(access)
            state.rights = access
        state.current = next
```

So, the full lookup without caching is as shown below. For efficiency, the implementation will check the accumulated global rights calculated by the parser before performing a tree lookup. There are probably many cases where the result of any lookup can be predicted from those rights, and therefore the lookup itself, and even creation of the filtered tree, could be skipped entirely.

```
    lookup(tree : ref to filtered-tree, path : string):
        state = init(tree)
        foreach segment in path
            if done(state)
                break;
            state = step(state, segment)
        return state.rights
```

Better Access Control

TODO after 1.10.

Implementation Notes

Changes from Current Behaviour

Rules Defined Only Once

The config-based parser would merge and override access entries in redefined rules. The new parser rejects authz files that define the same rule more than once.

New Rule Syntax for Wildcard Rules

Existing rules come in two flavours; repository-specific and global:

```
[repos:/path]
[/path]
```

In these rules, `/path` is always matched literally. The new parser supports two new forms for rules with paths that contain wildcards:

```
[:glob:repos:/path]
[:glob:/path]
```

Because a glob rule is not required to actually contain wildcards in the path, two sections with different names may represent the same rule; for example,

```
[/path]
[:glob:/path]
```

the above two rules are identical. The new parser detects (and rejects) such collisions.

Write-only Access is Not Allowed

The config-based authz parser will allow access entries that grant only write access; e.g.,

```
[/]
* = w
```

The new parser flags such entries as invalid; our authz implementation does not support write-only access.

Only Group Definitions in the Global Groups File

The new parser allows only group definitions (i.e., only the `[groups]` section) in the global groups file. The config-based parser would allow other sections there, but they would be ignored.