Director's structure

I) implemented as a low level disk file, 0-level
   contains 2 major parts

A) allocation and accounting control

associated with certain directories is something like an ER's allocation block. This data will either be stored in some data words in the first part of the low level disk file, or will be pointed to by a word in first part of the low level file. Each directory not having an associated "allocation block" will point to its 1st ancestor that does. Thus each directory will have an indirectly associated "allocation block". Any object owned by a directory will have its space requirements paid for by the allocation block either directly or indirectly associated with that directory. This allocation block will be pointed to by the object itself.

(Not: The ancestor of a directory is the directory owning the given directory)

example

```
<table>
<thead>
<tr>
<th>directory</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 object (other than directory)</td>
</tr>
<tr>
<td>1 allocation block</td>
</tr>
<tr>
<td>→ ownership pointer</td>
</tr>
<tr>
<td>--→ containing allocation block</td>
</tr>
</tbody>
</table>
```

B) directory body

contains names and access control for various objects, i.e. "entries"
II) directory entries

Each entry contains a number of parts.

A) **name**

Text name of this entry. The format of the name as well as the algorithm for finding an entry for a given input name are in dispute.

Some possibilities are:

1. **The name is a sequence of printable characters.**
   (There may be a fixed maximum length, or arbitrary length)

   Recognition could be by either exact match (A)
   or the unique entry the 1st part of whose name
   exactly matches the given input name (B)
   or the 1st entry the 1st part of whose name exactly
   matches the given input name (C)

Examples

<table>
<thead>
<tr>
<th>Entries</th>
<th>Entries</th>
</tr>
</thead>
<tbody>
<tr>
<td>directory 1</td>
<td>alpha 1</td>
</tr>
<tr>
<td>beta</td>
<td>alpha</td>
</tr>
</tbody>
</table>

[Signature]

[Date]
in directory 1 under schemes a), b), c) the following given inputs get the entries as shown

a)  

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>no match</th>
<th>no match</th>
<th>alpha1</th>
</tr>
</thead>
<tbody>
<tr>
<td>alpha</td>
<td>alpha</td>
<td>no match</td>
<td>alpha1</td>
<td></td>
</tr>
<tr>
<td>alpha1</td>
<td>alpha1</td>
<td>alpha1</td>
<td>alpha1</td>
<td></td>
</tr>
</tbody>
</table>

b)  

<table>
<thead>
<tr>
<th></th>
<th>alpha</th>
<th>alpha1</th>
</tr>
</thead>
<tbody>
<tr>
<td>alpha</td>
<td>alpha1</td>
<td>alpha1</td>
</tr>
<tr>
<td>alpha1</td>
<td>alpha1</td>
<td>alpha1</td>
</tr>
</tbody>
</table>

If there is an exact match, note that if no exact match, if there is a unique entry whose 1st part exactly matches the given input name, note that otherwise no match.

d)  

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>no match</th>
</tr>
</thead>
<tbody>
<tr>
<td>alpha</td>
<td>alpha</td>
<td></td>
</tr>
<tr>
<td>alpha1</td>
<td>alpha1</td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>beta</td>
<td></td>
</tr>
</tbody>
</table>

The difficulty with schemes that use initial abbreviation (schemes b), c) and d) above) is that frequently one uses the last part of a name to distinguish files, e.g. alpha, alpha1, alpha2, etc. In this case the abbreviation is of no help.
2) The name is divided into parts and possibly subparts. The parts are supposed to indicate what kind of object is being named, for example, with a 2 part naming system. The second part could be binary, fraction, compass, etc., to indicate that the file is binary input from a header or source input for fraction or compass.

Each part can now be abbreviated separately. An algorithm might be as follows:

```plaintext
construct a name parts by parts as follows:
for each part if the given input name part exactly matches some corresponding part in an entry, take that. else if there is some unique corresponding part in an entry an initial segment of which matches the given input name part, take that; otherwise no match.
```

Having constructed a name, if there is a unique entry which matches the constructed name, take it; otherwise no match.

The name can be divided into 2 major parts, each of which has subparts. Then corresponding parts are $i^{th}$ subpart of $j^{th}$ major part where $j = 10x2$ and $i = 1, 2, \ldots$
an attempted example follows, using the 2 major part scheme, major parts divided by colon, minor parts divided by period.

\begin{tabular}{ll}
  directory & BLAST:IA:Fortran \\
  & BLAST:IA:Binary \\
  & BLAST:2:Fortran \\
  & ALPHA:Fortran \\
  & BAKER:2:Binary \\
  \hline
  input & match \\
  B.1:F & (no match) \\
  B.1:I= & BLAST:IA:Fortran \\
  A.2:8 & BAKER:2:Binary \\
\end{tabular}

It would appear that the 1st example should have picked up "BLAST:IA:Fortran" as not is in some sense a unique match. But I don't know how to state an algorithm I would accept which would work this way.
b) **object**

This part of the entry comes in 3 flavors

1) **ownership**

for some kinds of objects there are no ownership entries, for others there is exactly one director with exactly one ownership entry for the object. (The later include at least disk files, directories, and subprocess descriptors)

An ownership entry includes some sort of unique designation of the object (for disk files, directories and subprocess descriptors this is the unique name and disk address.)

Further specification will be given when the various kinds of object are described.

2) **non ownership or hold link**

consists of the same unique designation that an ownership entry would contain.

3) **soft link**

consists of a) hard link to a directory

b) hard link to an accessory
c) text name

The obvious look up in the specific directory is made.
c) **access list**

This is a list (of arbitrary length) of pairs. The list member of each pair is an access key number. The 2nd member of each pair is the set of allowed options for the descendant object to be permitted using this accessor.

(This second member may also contain the type field of the object for convenience of implementation, but this is invisible to the user.)

D) **scratch bit**

One of the option bits associated with directory access is implicit access. A reference to an ownership type entry via a directory capability with the implicit access bit on is an implicit access key. The scratch bit is an implicit access key. If the scratch bit is off, the associated options permit all actions. If the scratch bit is on, they permit only destruction.