Neat Filesystem Traversal by DCG

Roy Ratcliffe[1,*]

Abstract

Explores an innovative approach to filesystem traversal using Definite-Clause Grammars (DCGs) in Prolog. Traditionally employed for parsing and language processing, the approach repurposes DCGs to navigate directory structures in a declarative and modular fashion. This method leverages Prolog's pattern matching and backtracking strengths to create concise and expressive traversal logic. The approach facilitates clear and maintainable code for filesystem operations by utilising DCGs, offering advantages over conventional imperative methods.

Keywords: Prolog, DCG

Sometimes it is advantageous to recursively traverse files and sub-files within a directory tree by *partial unification* using definite-clause grammars, i.e. difference lists.

It can be helpful to extract file system information in two forms:

i. the sub-file path *and also*
ii. the components of that path.

This is what's required: a grammar of the form

$$directory\_entry(+Directory, -Entry)//$$

which recursively unifies full paths of files at *Entry*, where the difference list unifies with the recursive path components of each found file.

1. Solution by DCG

In Prolog DCG form, the solution appears below. The first *directory_entry*//2 predicate works on the difference list. It neatly traverses a file system using a grammar.

```
%!  directory_entry(+Directory, ?Entry)// is nondet.
%
%   Neatly traverses a file system using a grammar.

directory_entry(Directory, Entry) -->
    { exists_directory(Directory),
      !,
      directory_entry(Directory, Entry_),
      entries_entry([Directory, Entry_], Directory_)
```

*Corresponding author
Email address: roy@ratcliffe.me (Roy Ratcliffe)
[1]See more hackery at GitHub.

```
    },
    [Entry_],
    directory_entry(Directory_, Entry).
directory_entry(Directory, Entry, [], Entries) :-
    entries_entry([Directory|Entries], Entry).

entries_entry(Entries, Entry) :- atomic_list_concat(Entries, /, Entry).

%!  directory_entry(+Directory, ?Entry) is nondet.
%
%   No need to check if the Entry exists. It does exist at the time of
%   directory iteration. That could easily change by deleting, moving or
%   renaming the entry.

directory_entry(Directory, Entry) :-
    directory_files(Directory, Entries),
    member(Entry, Entries),
    \+ special(Entry).

special(.).
special(..).
```

The implementation depends on `directory_files/2`, which finds a list of *Entries* within a given *Directory*. Atoms make up the list with their case preserved.

The given solution only finds files and skips the special dot entries. Here, *Entry* refers to a file. The grammar recursively traverses sub-directories beneath the given *Directory* and yields every existing file path at *Entry*. The directory acts as the root of the scan; it joins with the entry to yield the full path of the file, but **not** with the difference list. The second *List* argument of *phrase*/2 unifies with a list of the corresponding sub-path components *without* the root. The caller sees the full path *and* the relative sub-components.

Note that the second clause appears in the DCG expanded form with the two hidden arguments: the pre-parsed input list S_0 and the post-parsed output list S. For non-directory entries, the input list unifies with nil `[]` because it represents a terminal node in the directory tree, and the post-parsed terms amount to the accumulated *Entries* spanning the sub-directory entries in-between the original root directory and the file itself.

1.1. Why is this useful?

Good question. In the Prolog world, this approach allows for selective unification based on *relative file path components*. Take an example.

The following `phrase/2` finds all the files in the root directory of drive C.

```
?- phrase(directory_entry('c:/', A), [B]).
A = 'c://hiberfil.sys',
B = 'hiberfil.sys' ;
A = 'c://pagefile.sys',
B = 'pagefile.sys' ;
A = 'c://swapfile.sys',
B = 'swapfile.sys' ;
false.
```

In short, this becomes useful for filtering file system traversals nondeterministically. Use variables for indeterminate components of the path and atoms for determined components.

As another example, take the following. It finds all files and matches the last *Entry* term, i.e. the file's name within its parent directory, and matches that atom by file name extension. This takes longer to execute, naturally.

```
?- phrase(directory_entry('c:/', A), B), last(B, C), file_name_extension(D, ini, C).
```

The result in *A* and *D* is the full path name of the file and its base name, respectively.

2. Conclusions

The implementation does not require an `append/3`. The difference list implicitly walks the sub-directory tree and automatically appends the list with the terminal elements, in the case of the non-directory entries found within each directory.

Of course, performance is a consideration. The grammar does **not** automatically descend the entire directory tree. It descends the tree as far as the difference list requires, but no more. The length of the list prescribes the bounds of the search and limits the depth.

The solution is a handy shortcut for partial searching by path components by an arbitrary mixture of knowns and unknowns. One might describe it as a "neat" and somewhat creative use of definite-clause grammars.