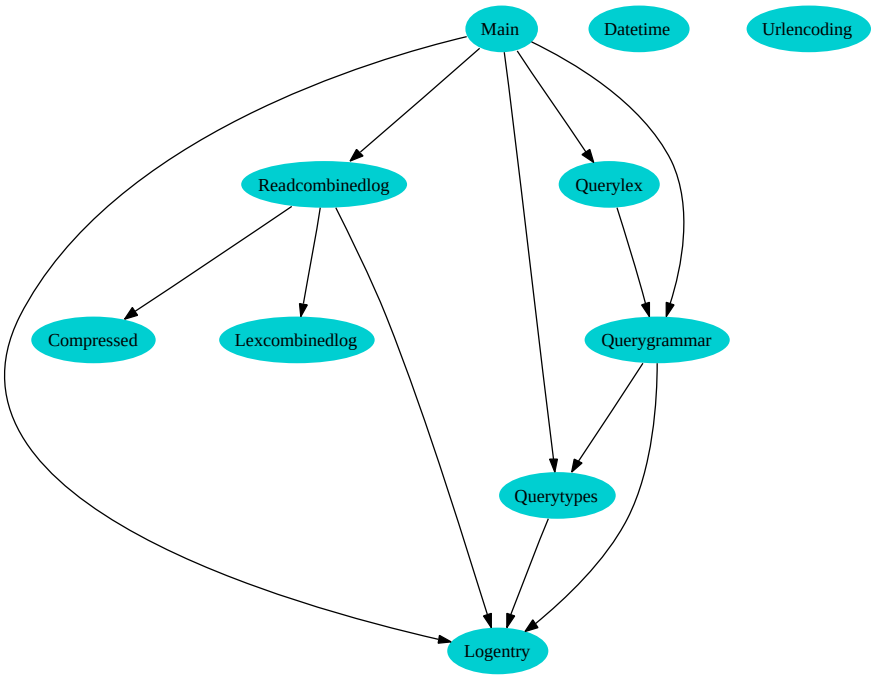
# Design Overview on apalogretrieve

Oliver Bandel

31. Januar 2008

# 1 The used compilation units




| | |
|---|---|
| logentry.ml | typedefinitions of the Log-Entry-Records |
| logentry.mli | Interface definition for `logentry.ml` |
| lexcombinedlog.mll | Lexer definition for Logfiles |
| main.ml | Main part of apalogretrieve |
| querytypes.ml | definition of the result-types of the query-parser |
| readcombinedlog.ml | the reader-functions for the Apache Combined Logfile |
| querygrammar.mly | the granmmar definition for the SQL-parsing |
| querylex.mll | the scanner for the SQL-parsing |

## 1.1 Main

`Main` contains the REPL-loop for the SQL-like queries. For each complete query the Logfile will be read.

There is no caching of the already read information. For small logfiles this is ok. For

big logfiles a caching-mechanism might be implemented, if necessary (if many queries will be done).

## 1.2 Loglex

`Loglex` is a simple scanner that only grabs the next information in a logfile out and gives it back as a string. `Loglex` does not check the syntax on a higher level. It only separates text from `[` and `]` or `"` and `"`.

```
hostname.com - - [Date-and-Time] "GET /robots.txt HTTP/1.1" 404 216 "-" "Browsername"
hostname2.au - - [Date-and-Time] "GET /software/ HTTP/1.1" 200 1685 "-" "Another Browsername"
```

## 1.3 Logentry

`Logentry` contains functions to create Entry-records as well as functions to retrieve items from the record as well as a function that extracts all items of a query from a record.

`Logentry` is handling the logentries in the most abstract way: it is a helper for other modules in the project and determines, which values the different logfile-scanners has to provide.

## 1.4 Readcombinedlog

`Readcombinedlog` reads a logfile that is in the „Apache Combined Logfile Format“. It is a kind of high-level parsing, which uses Loglex for the logfile-scanning.

On page 3 you can see an (schematic) example of a logfile in the *Apache Combined Logfile Format*. It has the following structure:

1. Hostname
2. ??? m1
3. ??? m2
4. Date, Time and possibly Timeshift for timezone (e.g. `+0100`)
5. HTTP-Request
6. Returncode for the Request (Status)
7. Size of the requested document
8. Referrer (last visited page before using the link to the requested page)
9. Client-Name (e.g. Browser-name)

## 1.5 Querytypes

`Querytypes` contains type-definitions that are necessary for the handling of queries.

## 1.6 Querylex

`Querylex` is the scanner for the terminal symbols that are used in the SQL-like statements (read in the REPL-loop), which the user types in (or pipes in).

## 1.7 Querygrammar

`Querygrammar` is the implementation of the grammar of the SQL-like statements.

Here the query-syntax is implemented.

`Querygrammar` takes the tokens from `Querylex` and puts together a datastructure that contains the information, that are used by the `query`-function in the module `Main`.

The filters for the `WHERE`-clauses are composed of partial applicated functions (closures). For complex queries, the already created closures will be composed together again.

### 1.7.1 Short overview on constructing result values of type query_t

Here you can see the type of the resultvalue of the query-parser.

```
type query_t = { sel: Logentry.entry_type_t list;
                 file: string;
                 cond: (Logentry.entry_t -> bool) option }
```

As you can see, there is a list of `sel`ections, the filename and a `cond`ition-value with type `(Logentry.entry_t -> bool) option`.

The optional `cond`ition value is not a list. Even if you have a lot of selection-criteria (`WHERE`-clauses) there only is one value that will optional (only if there is at least one condition) be used as a filter.

How is this filter being created?

*This arrow starts to be a part of a graphic that explains the filter-creation (partial application). This arrow shows me, that tikz works fine ;-))*