UNIVERSITY OF OSLO
Faculty of Humanities
Department of Linguistics and
Scandinavian Studies

# Rapid parser prototyping in Common Lisp

## Thesis for the cand.philol. degree

Christian Nybø
University of Oslo

November 30, 2005

# Abstract

The syntax and semantics of internet standards are most often specified in an Internet Request for Comments (RFC). The specification for the Hypertext Transfer Protocol (HTTP) (RFC 2616) presents the grammar for its messages in a form known as Backus-Naur form (BNF). The application reads these statements to generate a parser that reads http messages. To demonstrate this technique, we use the parser as a building block in a web server.

# Contents

0.1 Thanks . . . . . . . . . . 6

**1 Introduction 7**

1.1 Choice of topic . . . . . . . . . . 7
1.2 Other implementations . . . . . . . . . . 8
1.3 Pragmatism crowds out ideal solutions . . . . . . . . . . 9
1.4 From a mouse click to a web page . . . . . . . . . . 9
1.5 How HTTP and other net standards are specified . . . . . . . . . . 10
1.6 What is new about HTTP 1.1 . . . . . . . . . . 10
1.7 Why performance matters . . . . . . . . . . 11

**2 Components of the system 13**

2.1 Compiling and loading the components . . . . . . . . . . 14
2.2 Simple abstractions . . . . . . . . . . 15
2.3 Date and time . . . . . . . . . . 15
2.4 A language implementation . . . . . . . . . . 15
2.5 Compatibility with other implementations . . . . . . . . . . 16
2.6 Virtual hosts . . . . . . . . . . 16
2.7 Logging . . . . . . . . . . 17
  2.7.1 Using the syslog facility . . . . . . . . . . 17

**3 The request 19**

3.1 Backus-Naur form . . . . . . . . . . 21
3.2 The META language . . . . . . . . . . 22
  3.2.1 Specifying comma-separated lists . . . . . . . . . . 25
3.3 Match a type rather than characters . . . . . . . . . . 26
3.4 Parse tree . . . . . . . . . . 31
3.5 Pragmatic parsing . . . . . . . . . . 31
  3.5.1 Intentional capture with special variables . . . . . . . . . . 31
  3.5.2 Keeping the global namespace clean . . . . . . . . . . 33
  3.5.3 Using grammar from another RFC . . . . . . . . . . 33
  3.5.4 Specifying the format of the MIME file in BNF . . . . . . . . . . 36

3.6 Checking for type definitions . . . . . . . . . . . . . . . . . . . . . . 37
3.7 Ignoring whitespace . . . . . . . . . . . . . . . . . . . . . . . . . . . 37
3.8 From a flat parse to a parse tree . . . . . . . . . . . . . . . . . . . 38
3.9 Memoization . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 41

**4 The response** **42**
4.1 The resource . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 42
4.2 The variant . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 42
4.3 The entity . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 42
4.4 Semantics of HTTP . . . . . . . . . . . . . . . . . . . . . . . . . . . . 42
  4.4.1 A language for semantics? . . . . . . . . . . . . . . . . . . 43
4.5 Sending the response . . . . . . . . . . . . . . . . . . . . . . . . . . 43
4.6 Speeding up the file transfer . . . . . . . . . . . . . . . . . . . . . 44
4.7 MD5 . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 45

**5 The server** **46**
5.1 Multiple processes . . . . . . . . . . . . . . . . . . . . . . . . . . . . 46
5.2 Signal handling . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 47
5.3 Error handling . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 47
5.4 Profiling . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 47
5.5 Dumping the application . . . . . . . . . . . . . . . . . . . . . . . . 48

**6 Further work** **49**
6.1 Methods and combinations . . . . . . . . . . . . . . . . . . . . . . . . 49
  6.1.1 Methods . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 49
  6.1.2 Conditional requests . . . . . . . . . . . . . . . . . . . . . . 49
  6.1.3 Ranges . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 50
  6.1.4 Content negotiation . . . . . . . . . . . . . . . . . . . . . . . 50
6.2 Session handling . . . . . . . . . . . . . . . . . . . . . . . . . . . . 50
  6.2.1 Cookies . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 51
  6.2.2 URL rewriting . . . . . . . . . . . . . . . . . . . . . . . . . . . 51
6.3 Web server stack . . . . . . . . . . . . . . . . . . . . . . . . . . . . 51
  6.3.1 Operating system . . . . . . . . . . . . . . . . . . . . . . . . . 51
  6.3.2 Database server . . . . . . . . . . . . . . . . . . . . . . . . . . 52
  6.3.3 Server-side scripting language . . . . . . . . . . . . . . . . . 52
  6.3.4 Authentication . . . . . . . . . . . . . . . . . . . . . . . . . . 53
  6.3.5 Secure Sockets support . . . . . . . . . . . . . . . . . . . . . . 53
  6.3.6 Load balancing . . . . . . . . . . . . . . . . . . . . . . . . . . 53

**7 Conclusion** **54**

**A Colophon** **59**

**B Acronyms** **60**

**C Source code** **62**

C.1 Line counts . . . . . . . . . . . . . . . . . . . . . . . . . . . . 62
C.2 Call graphs . . . . . . . . . . . . . . . . . . . . . . . . . . . . 62
C.3 Utilities . . . . . . . . . . . . . . . . . . . . . . . . . . . . 67
C.4 Meta . . . . . . . . . . . . . . . . . . . . . . . . . . . . 72
C.5 BNF . . . . . . . . . . . . . . . . . . . . . . . . . . . . 76
C.6 http . . . . . . . . . . . . . . . . . . . . . . . . . . . . 90
C.6.1 Configuration . . . . . . . . . . . . . . . . . . . . 90
C.6.2 Formatted strings . . . . . . . . . . . . . . . . . . . . 90
C.6.3 Utility functions . . . . . . . . . . . . . . . . . . . . 91
C.6.4 URI . . . . . . . . . . . . . . . . . . . . 91
C.6.5 Packages for rulesets . . . . . . . . . . . . . . . . . . . . 92
C.6.6 Presets for ruleset . . . . . . . . . . . . . . . . . . . . 92
C.7 date-parser . . . . . . . . . . . . . . . . . . . . . . . . . . . . 114
C.8 md5 . . . . . . . . . . . . . . . . . . . . . . . . . . . . 117
C.9 server-process . . . . . . . . . . . . . . . . . . . . . . . . . . . . 121
C.10 clack.asd . . . . . . . . . . . . . . . . . . . . . . . . . . . . 123
C.11 sample-rules.txt . . . . . . . . . . . . . . . . . . . . . . . . . . . . 124
C.12 mime-filter . . . . . . . . . . . . . . . . . . . . . . . . . . . . 126

## 0.1 Thanks

I'd like to thank the Norwegian Embassy in Lusaka as well as Ms. Karen Jørgensen of Chifwema Road for borrowing me office space while in Zambia.

Herman Ruge Jervell, my thesis advisor, helped me out when I started the work on this thesis. As I have lived in Lusaka, Zambia during this period, I did the coding mostly on my own, and he has advised me the last four months of the project, after I came back to Oslo.

# Chapter 1

# Introduction

## 1.1 Choice of topic

I have added parsing functionality to Common Lisp. The editable readtable of Common Lisp lets us adapt the reader to syntaxes that are similar to that of Common Lisp. For other input formats, altering the readtable is too hairy for most programmers. This package will make it easy to read foreign syntaxes as long as a BNF specification of the syntax is available.

eXtensible Markup Language (XML) is popular, both as a data interchange format between programs, and increasingly as a format for information intended for people to read. With XML as the input format and an appropriate XML input library, the scanning part of the parsing process is no longer an issue, but translating from the format you get to the format you want, is still a task. In other parts of the language landscape, transformations using eXtensible stylesheet language (XSL) will be the way from a parsed version of the input, while when writing a program in a language with s-expressions, such as lisp, we will probably prefer to do these transformations in the language proper.

As an application of this technique, I have implemented a web server, thus showing that there is a considerable gain in letting the software generate the parser rather than writing it oneself. The macro `with-bnf` expands to cases covering all of the six hundred symbols used in the HTTP specification. This would have been error-prone to do by hand. `with-bnf` constitutes a virtual laboratory for trying out various techniques in parsing the messages, as we can alter the macro and rather instantly see the impact of the change for all the symbols and their grammar rules.

The third version of the hypertext transfer protocol, HTTP 1.1, offers opportunities to save bandwidth and decrease response time. As the mechanism for virtual hosts provides significant savings for the internet services providers, there are few servers running earlier versions than HTTP 1.1.

As many web applications get fairly close to the protocol, it is at this juncture up to the application programmer rather than the author of the web server to realize these savings.

It is more work to implement a server-side executed page that correctly implements performance-enhancing features of the protocol such as *if-modified--since*, *last-modified*, compression, and *content-length*, than merely replying as if the client had made an unqualified GET request.

Improvements on the server side have the most impact for clients that suffer from large latency or narrow bandwidth when talking to the server. As these input variables may vary, it is a little difficult to precisely quantify the savings generated for a single page, and as development takes place in environments provided with seemingly unlimited bandwidth and low latency, it is tempting to take shortcuts. These shortcuts in programming may add up to sluggish performance for users far away from the server. Better knowledge of the HTTP protocol combined with dedication to performance may result in a satisfying user experience even with high latency and low bandwidth.

Non-industrialized regions will for some time be constrained to the performance of satellite connections through Very Small Aperture Terminals (VSATs), phone lines or GSM [Sarrocco, 2002]. Websites using web server applications that minimize bandwidth requirements will be attractive to these regions. Let's first have a look at what other web servers there are.

## 1.2 Other implementations

The industry standard for web servers is the Apache webserver [The Apache Group, 2005], whose code base descends from the NCSA webserver [NCSA HTTPd Development Team, 1997], written by Rob McCool. Within the Common Lisp community, the first initiative was John C. Mallery's CL-HTTP [Mallery, 1994]. The Lisp implementation company Franz Inc. have made their AllegroServe software [Foderaro, 1999] available in source code form, and others have ported it to other lisp implementations under the name Portable Allegroserve.

Web server projects have different aims. Apache started out a set of patches to the NCSA server, and as such, had pragmatic goal right from the start. The AllegroServe project was launched in order to provide an example of network programming in Common Lisp. Pragmatism is a core trait of both TCP/IP, WWW and the Apache web server.

## 1.3 Pragmatism crowds out ideal solutions

The concept of hypertext is not new. Neither is the idea of distributed hyperlinked information systems. What Tim Berners-Lee did when he invented World Wide Web at European Organization for Nuclear Research (CERN) [Gillies and Cailliau, 2000] was adding pragmatism to the mix. Rather than designing a beautiful and complete system, which is what Ted Nelson did with Xanadu [Gillies and Cailliau, 2000], Berners-Lee implemented a system with a few improvements compared to the then-dominantGopher service [Anklesaria et al., 1993], and let it loose. Marc Andreesen, a student at The National Center for Supercomputing Applications (NCSA), let the browser display images within the text of pages [Andreesen, 1993], and suddenly a new visual medium had been born. Such is the story of the mishmash of rewritten standards that make up the infrastructure of the web, in that standards have been written, implemented and refined in several iterations. The winning algorithm is to produce something that gathers a user-base, and when there is a user-base, the user-base will improve on the product.

Xanadu compared to world wide web is like OSI compared to internet - one is specification first, aiming for perfection, while the other is "rough consensus and running code", creating something of some value, letting it loose, and improving as one goes along.

There are rough edges to the specification of HTTP, and the standard itself is not as well-written as it could have been if it were written by a single author with precognition of todays demands. Still, it has had tremendous impact, and we will have to deal with these rough edges. But I will start with the broader picture.

## 1.4 From a mouse click to a web page

Before we get down to the details, we will have a look at how web servers fit into the World Wide Web infrastructure, from the viewpoint of a user clicking a link on a web page.

A link has an action associated with it, described by the protocol element of the Uniform Resource Locator (URL). Most often the protocol is HTTP, and the corresponding action is to fetch a document. The location of the document is described by a uniform resource locator, a URL. When a link is clicked, the browser checks to ensure that it has a connection to the server of the url, looks for the expiry date of a local copy, and asks the server for a newer copy if there is one. If so, the new copy is downloaded, if not, the local copy is used. The browser looks through the copy to see if there are other resources that may be needed when displaying the copy.

In between the origin server and the client, there may be proxies. These maintain a cache of previously requested resources, tagged with timestamps describing when they were generated, fetched and when they expire. Each cache verifies that their copy is as new as the version on the next server in the chain. In this project, I intend to write an origin server, therefore I will mostly disregard the cache functionality.

With a sketch of the pragmatics of HTTP, the formal part of the process remains to be described.

## 1.5 How HTTP and other net standards are specified

There is a need or an opportunity for a standardization, and someone writes a specification. This someone may be a lone hacker, an employee at a company, an academic, or someone who represents a blend of these roles.

The specification is then distributed as an Internet Draft, and after peer review, its may be promoted to a Request for Comments. The name is somewhat misleading, as the comments had better be in before the document reaches status of an RFC [Bradner, 1996].

The current version of HTTP, HTTP 1.1, is specified in the internet RFC 2616 [Fielding et al., 1999].

## 1.6 What is new about HTTP 1.1

When a web client talks directly to a web server, there is a Transfer Control Protocol / Internet Protocol (TCP/IP) connection between the two. In early versions of HTTP, the connection would be closed once the client got the answer to its request. In HTTP 1.1, the TCP/IP connection is kept alive. With the *keep-alive* directive introduced experimentally with HTTP 1.0 [Fielding et al., 1997], the client does not have to close the connection, as the server will keep in listening for new requests.

This technique is sometimes referred to as pipelining. In addition to pipelining, some browsers understand hints on prefetching, that is, a page may tell the browser about other pages that are likely to be the next page asked for by the user. The browser may then download this page in advance, and present it immediately in the likely case the user asks for that page [Fisher, 2003]. With prefetching, the net result may be that more bandwidth is spent, but to the user, there may be a gain, as the next page may be available without any noticeable lag. Authors and programmers of web systems are at liberty to ignore the potential improvements offered by HTTP 1.1. Rather than alienating users that are connected with high latency and/or low bandwidth, authors, programmers and adminstrators may spend the extra time it takes to take advantage of HTTP 1.1 to make pages and websites respond faster.

## 1.7 Why performance matters

Moore's law describes the development of computing power, in that it doubles every 18 months. With Moore's law, is there any point in being concerned with the performance of software? Even for software running in a single computer, Moore's law may in itself run into physical limitations, but in our case, we are concerned with the performace of networked applications, and therefore we must consider the *connections between* computers in terms of bandwidth and latency. Ubiquitous bandwidth is still not present. The digital divide will probably be bridged by powerful cellphones, but it is costly to build networks that offer bandwidth above that needed for speech. Without multiplexing, Global System for Mobile Communications (GSM) offers 9600 bits per second. The more efficient our use of the network, the more people can connect to it and get useful responses within their budget. Large areas of the world are only serviced through satellite connections, and as the satellites are far away from the terminals, the latency of half a second is not likely to go away.

Jacob Nielsen [Nielsen, 1997] talks of the bandwidth from his home to the internet expanding at 60% a year.

Complex internet applications are attempts at hiding the distance between the user and where the actual application is running. Physics dictate that this gap is unlikely to close soon.

When a personal computer is running at clock speed of 2 gigahertz (GHz), light only travels 15 centimeter per clock cycle:

```
BNF> (format nil "Light travels ~F cm per clock cycle"
     (let* ((ghz (expt 10 9))
    (light 299792)
    (light-in-cm (* light 1000 100)))
       (/ light-in-cm (* 2 ghz))))

"Light travels 14.9896 cm per clock cycle"
BNF>
```

As light moves at 299792 kilometers per second, and a modern pc goes through 2 billion clock cycles per second, light can only move less than 15 centimeters per clock cycle. Therefore, we can build faster computers, and get more bandwidth, but network latency has an upper bound, and will remain the bottleneck for web applications.

Given that that knowledge of HTTP is useful, we will move on to an implementation.

# Chapter 2

# Components of the system

A web server has to make sense of requests in order to respond to them. The format of the requests is specified in BNF expressions littered around in the HTTP specification. I have assembled these in the file `sample-rules.txt`. The format is straight-forward, although rule definitions may be broken over several lines. If the first non-whitespace character of the next line is placed beyond the column of the equal sign, this line is part of the rule definition of the previous line.

The file `bnf.lisp` teaches the lisp to read these BNF expressions, and as they are read, they are translated to META, a small language built on top of lisp, then converted to lisp code. This lisp code is the parser for HTTP messages.

Once we know how to parse HTTP messages, the file `http.lisp` contains the procedures to receive and respond to single requests in http. The file `server-process.lisp` adds two abilities - the ability to respond to another request once one is replied to without closing the socket, and the ability to have conversations with multiple clients at the same time.

There are another two source files, `date-parser.lisp` and `md5.lisp`, they are both parts of `http.lisp` that are large enough to warrant separate files. `date-parser` reads the date formats of HTTP and converts them to number of seconds since 1900, while md5 provides a basic implementation of a checksum algorithm used to ensure that a file is intact once it is received at the endpoint.

I will demonstrate the use of BNF for prototyping by writing a simple web server. The server comprises of the following parts:

**Utilities** are syntactic sugar in the form of macros and small functions.

**Meta** provides a reader for the META language, as described in [Baker, 1991].

**Bnf** extends the system with another small language, to generate parser code from a collection of BNF statements.

**Http** uses bnf to generate a parser, and provides the semantics of an http dialogue. Two files supply functions to the http file:

  **Date-parser** reads and formats datetimes.

  **Md5** calculates an MD5 checksum for a string or a file

**Server-process** sets up a process that listens to a port and dispatches requests to new threads.

Together these components provide the essentials of a web server, that is, one that serves pages, but does not adhere to the full set of requirements outlined in the HTTP RFC.

## 2.1 Compiling and loading the components

There are comprehensive facilities for loading and updating parts of a system. Although not part of the language proper, Mark Kantrowitz' *defsystem* [Kantrowitz, 1991] has been the de-facto standard for a long time. A recent replacement, Another System Definition Facility (asdf) [Barlow, 2004], is adapted to the need to download system requirements on-the-fly, if needed. So far, we have resorted to a simple solution; a file `def.lisp` that sets up the default path, compiles, and then loads the parts of the system.

Starting the system from a bare lisp consists of these three steps:

```
(cwd ‘‘/Users/chr/proj/uio/sli/hovedfag’’)
(load ‘‘def’’)
(server:start)
```

After these steps, the server will listen to and answer requests on, port 8000.

With the asdf facilities, a system definition, contained in the file `clack.asd` looks like this:

```
(defsystem "clack"
    :description "clack: a web server."
    :version "1.0"
    :author "Christian Nybø <chr@sli.uio.no>"
    :licence "Public Domain"
    :components ((:file "utilities")
(:file "meta" :depends-on ("utilities"))
(:file "bnf" :depends-on ("meta"))
(:file "http" :depends-on ("bnf"))
(:file "server-process" :depends-on ("http"))))
```

and to load it with the asdf system present, you would say:

```
CL-USER> (asdf:operate 'asdf:load-op :clack)
```

## 2.2 Simple abstractions

The source file `utilities.lisp` contains various macro definitions that help shorten the code. I find that extending the language with simple rewriting macros to be a rather inexpensive addition, as readers unfamiliar with a particular macro may well macroexpand the form in the place where it was used, to see the effect of the macro. In that respect, a macro is almost self-explaining. On the other hand, a macro may well obfuscate complexity, in that only evaluating the macro may reveal what it does. One of my macros, that is, `with-bnf`, differs from the utility macros in that it macroexpands to thousands of lines of code [1], is quite complex, and it also depends on an input file.

## 2.3 Date and time

HTTP accepts time in three formats. As they are a little too readable for humans to be comfortable for computers, they are converted to universal times, that is the number of seconds since 1900.

## 2.4 A language implementation

I chose the open-source implementation *OpenMCL* [Byers, 2005]. Historically, it stems from Digitool's *Macintosh Common Lisp* .

Initial versions of the application used a few facilities from the `ccl-package`, that is, facilities that are idiosyncratic to the OpenMCL implementation. These forms were moved to a separate file, and instead there are implementation-dependent statements that require compatibility code. For the compatible forms needed for the file `http.lisp`, we look at `compat/IMPLEMENTATION-NAME/http.lisp`.

---

[1] the grammar of sample-rules expands to about 9680 lines of code, including debug statements

## 2.5 Compatibility with other implementations

Most of the server is written in plain Common Lisp. The exceptions are sockets, threads and the test for whether a symbol can be used to refer to a type. There are variants of writing a file to a stream that utilize OpenMCLs ability to call C functions, and those are also proprietary to the implementation. I will replace it with a slower but clean call to `write-sequence`.

I will remove `ccl` from the lists of imported packages, thus the namespaces will be free of references to OpenMCLs symbols, and rather set up a set of functions that a compatible system should provide.

`Server-process` sets up the server. That is about opening a socket, listening to it, and for each new connection, start a new thread that handles the requests on the connection. When there are no more requests, the thread should stop existing.

## 2.6 Virtual hosts

For a single physical server with a limited number of network interfaces to host many webservers with unique domain names, HTTP offers so-called *virtual hosting*. A single server has a single IP address, but there may be several domain names that point to this single Internet Protocol (IP) address. A single server answers HTTP requests to this server, but the requests are answered based on the Host:-header in the request. The Host:-header contains the domain name of the server as it is seen in the URL that the client is asking for.

The example server may be referred to as localhost or by its hostname “macchr”, and we want the server do act differently based on which of these names is used as the *host* part of the URL.

At this point, the server will serve pages also for urls that start with `http://macchr:8000`. The next step is making the server differ between hosts. A class `vhost` has a domainname, a document-root, a hashtable `functions` of mappings from url prefixes to functions, and a logfile.

The `request` class needs a new slot for the Host:-header, `receive-request` has to add it, and `servable-document-p` has to lookup an entity based on the Host: header. The variable `*document-root*` is kept as a fallback.

From the parsed request, we will probably know what the wanted “host” is set to, but as we don’t store the values from the parse as a hierarchical structure, we can’t tell whether a particular value for host is a result of parsing an `absoluteURI` or a Host:-header. Therefore it is neccessary to store not only a value for the header, but also store how we got to the header, the epistemology of the value, so to say. It is not sufficient to have the name of a host, we must also know where it comes from.

## 2.7 Logging

It's nice to know what the server is doing, and how much time it spends doing it. In logging, we are looking at the total time spent serving a page, rather to make sure that the end user does not have to wait for too long. If we are looking for performance bottlenecks, we should rather resort to profiling.

We start out with a function like this:

```
(defun http-log (request entity time code bytes-sent)
  (with-open-file (log "/tmp/log"
                       :direction :output
                       :if-exists :append
                       :if-does-not-exist :create)
    (format log "~A - - [~A] ~S ~D ~D~%"
            (remote-host request)
            (iso-8601-string)
            (request-line request)
            code
            bytes-sent)))
```

As we have several threads running at the same time, we need a way of ensuring that they do not write to the same file at once. On way is to ensure that no other threads attempt to write to the file while we write to the file. We use the implementation-dependent macro `(ccl:with-lock-grabbed ((ccl:make-lock "logfile"))` to make sure that different threads do not write at the same time.

A more advanced version would be to have a separate logger thread, and send messages from the other threads that ask the logger to write a line in the log. The logger could keep the file open, and the threads that wanted to write something to the log, could send a message to the logger and forget about it.

### 2.7.1 Using the syslog facility

I want the processes to output what they are doing. If they are to write to a common log file, there wil be the issue of locking the file, waiting for logs, picking a random time before retrying, and so forth. In unix, this has been solved by the *syslog* facilities. These are part of the C library. The call I want to use is

```
void syslog(int priority, const char *message, ...);
```

`int` is a number describing the severity of the condition, `*message` is a format string as to printf, and . . . are the argument(s) to the format string.

To use a C library, I'll need to create a C string. (ccl::make-cstring ”zot”) makes a C string. To call the C function syslog from openmcl, I can use the macro `#_syslog`, like this:

```
(#_syslog 6 (ccl::make-cstring "foo %s quux")
          :address (ccl::make-cstring "zot"))
```

In addition, I'll specify that everything from dppccl should be logged to a particular file clack.log, so I add the lines below to the unix configuration file `syslog.conf`,

```
!/Applications/openmcl/ccl/dppccl
*.* /var/log/clack.log
!*
```

(the first line says that the following statements concern those that are from the application `/Applications/openmcl/ccl/dppccl`, the last line resets this mode)

and restart `syslogd` with

```
killall -HUP syslogd
```

# Chapter 3

# The request

The request is what the client sends when asking for something.

The request specifies what the user wants. The answer depends on the following traits of the request:

| *Trait* | *sample value* | *Comment* |
|---|---|---|
| Method | GET | The method is the first element of the request-line |
| Host | www.uio.no | The host name of the web server |
| Port | 80 | A service of a unix server is associated with a port number |
| abs_path | /subjects/index.html | Path to the resource, which may or may not refer to a file on the server |

These decide what *resource* to look for, in short, this may be phrased as: `Method * Host * Port * abs_path → resource`, wherein the star signifies that the resource is the product of the combination of the four parameters.

There may be several representations of a resource, varying in each of these dimensions:

| | | |
|---|---|---|
| Accept | text/html, application/pdf | The preferred media-types of the client |
| Accept-Charset | iso-8859-1 | Preferred character set and encoding |
| Accept-Encoding | gzip, compress | Preferred compression scheme of the client |
| Accept-Language | en/en | Languages that the client wants to read |

In short, this may be phrased as: `media-type * Accept-Charset * Accept-Encoding * Accept-Language → variant` `media-type` is specified by the `Accept` header, but its name is undescriptive.

*How* to serve the answer depends on:

| | | |
|---|---|---|
| HTTP-Version | 1.1 | Does not change |
| Connection | Keep-Alive | An intention not to break the connection |
| Authorization | Basic QWxhZGRpbjp vcGVuIHNlc2FtZQ== | A token for allowing access |
| If-Match | "xyzzy" | A tag for a particular version of a resource or variant |
| If-Modified-Since | Sat, 29 Oct 1994 19:43:31 GMT | A date |
| If-None-Match | "xyzzy" | Also a tag |
| If-Range | Sat, 29 Oct 1994 19:43:31 GMT | Also a date |
| Range | bytes=0-499 | A set of byte ranges |

Again, this may be expressed as: `HTTP-Version * Connection * Authorization * If-Match * If-Modified-Since * If-None-Match * If-Range * Range` → `how to respond`.

There are headers that in some obscure cases may affect what and how to serve, such as `Referer` ("referer" has always been misspelled in HTTP) and `User-Agent`. They may have values such as http://www.uio.no/other.html and CERN-LineMode/2.15 libwww/2.17b3.

`Host`, `port`, and `abs_path` together produce an identifier for a resource (as a physical host can be pointed to by several hostnames, a process may listen to several ports.)

The first step is to identify the resource that the client wants, then whether it exists, if it is readable, and is allowed to be served to this user.

Once we have decided on a resource, we try to find the variant of the resource that is the most suitable for the client. After deciding on a variant, we'll look at the method used by the client, and answer according to it.

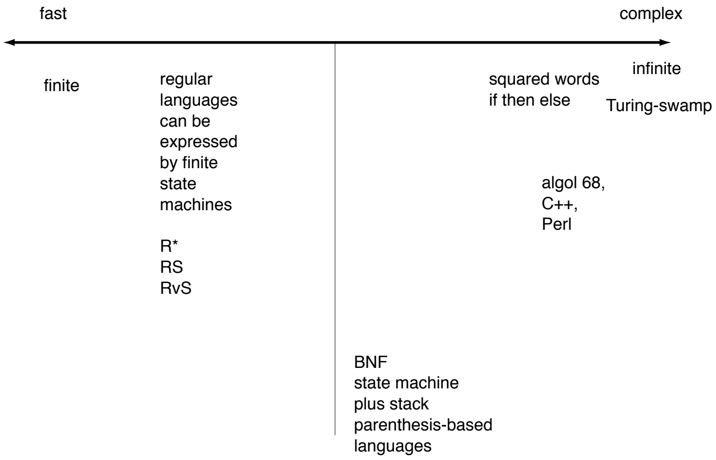
fast
complex
finite
regular languages can be expressed by finite state machines
R*
RS
RvS
BNF state machine plus stack parenthesis-based languages
squared words
if then else
infinite
Turing-swamp
algol 68, C++, Perl

Figure 3.1: The balance between decidability and expressiveness

## 3.1 Backus-Naur form

Lars Marius Garshol [Garshol, 1998] has written an informal introduction to Backus-Naur form (BNF).

BNF is used to specify a grammar for a context-free programming language or format. A grammar is a set of production rules that specify a set of symbols and which combinations of symbols are valid expressions in the language. If every production rule is the same regardless of what surrounds the occurence of the rule, we call it a context-free grammar. As BNF has no memory, and thus lacks facilities to specify the context in which a production rule occurs, the programming language needs to be context-free in order to specify it with BNF.

For example, in BNF we can not specify the property of well-formed XML that the start tag should have a matching end-tag. This must be specified by other means.

BNF represents a suitable balance between expressiveness and decidability, as illustrated in figure 3.1.

A language specified in BNF may be recognized by a finite state automaton with a stack.

The grammar of HTTP consists of a set of production rules. A production rule consists of a left-hand side, an equal sign and a right-hand side. The left-hand side specifies the name of the rule. The name is case sensitive as some names differ only by case, such as “Host” and “host”. The right-hand is the definition of the rule, and specifies what sequences are valid instances of the rule.

A definition is implicitly a sequence.

The first line in an HTTP request looks something like this:

`GET / HTTP/1.1`

The first lines of the BNF specifying the format of the first line an HTTP request:

```
Request-Line = Method SP Request-URI SP HTTP-Version CRLF
Method       = "OPTIONS"                ; Section 9.2
               | "GET"                  ; Section 9.3
```

The BNF format used in the HTTP specification is described in chapter 2, Notational Conventions and Generic Grammar in the RFC.

## 3.2 The META language

Henry Baker's [Baker, 1991] implementation of the *META* language is a small language built on top of Lisp. It adds matching constructs such as concatenation, alternation and kleene-star. Using meta as an intermediate format is not strictly necessary, we could well translate directly from BNF to lisp code, but having meta as an intermediate step makes for a compact, but readable representation of the rules specified in the BNF. Should macroexpansion and compilation speed be an issue, one could translate directly from BNF to lisp code. As we need to know how we drew some of the conclusions from a parsed request, our implementation of meta differs from the one sketched by Baker in that it builds a parse tree in addition to binding names to the result of applying the rule of the name.

The parser uses backtracking, parses from the top of grammar, parses from the left, and generates a leftmost parsing tree. As the parser works with a buffer and there is no effort to minimize the lookahead, it is not meaningful to qualify it with a value for how many characters it needs to look at. We may therefore characterize the parser as an LL parser.

The lisp syntax is extended with the following constructs:

| | |
|---|---|
| [ ] | surrounds a sequence |
| { } | surrounds a list of alternatives |
| ( ) | surrounds a grouping |
| @(type var) | match an element of type TYPE, bind the match to variable VAR |
| $ | Kleene star – match as many as possible of the following element |
| ! | escape character – evaluate the following lisp expression |

As the meta language extends the syntax of lisp, we have added a few characters to the syntax table of emacs, mimicking the syntax rules of parentheses. When emacs reads a file that ends with lines such as those below, it may evaluate those forms to adapt to the syntax of the file.

```
;;; local variables:
;;; eval: (modify-syntax-entry ?{  "(}  ")
;;; eval: (modify-syntax-entry ?}  "){  ")
;;; eval: (modify-syntax-entry ?\[ "(]  ")
;;; eval: (modify-syntax-entry ?\] ")[  ")
;;; end:
```

Below is a sample showing how to get from a fictious BNF statement as they appear in the specification, to an expression in the META language:

```
BNF> (parse-rule-definition "Method  = \"GET\" | \"HEAD\"
")
(RULENAME |Method| (("GET") (OR-TOKEN) ("HEAD")))
25
BNF> (normalize-rule *)
(RULENAME |Method| (PARSED-OR-EXPRESSION "GET" "HEAD"))
BNF> (bnf-expr-to-meta  (cddr *))
"{\"GET\" \"HEAD\"}"
BNF> (let* ((index 0)
    (string "GET")
    (end (length string)))
       (matchit #.(read-from-string *)))
(#\G #\E #\T)
BNF>
```

Figure 3.2 displays the steps from BNF expressions as found in the RFC to a parser.

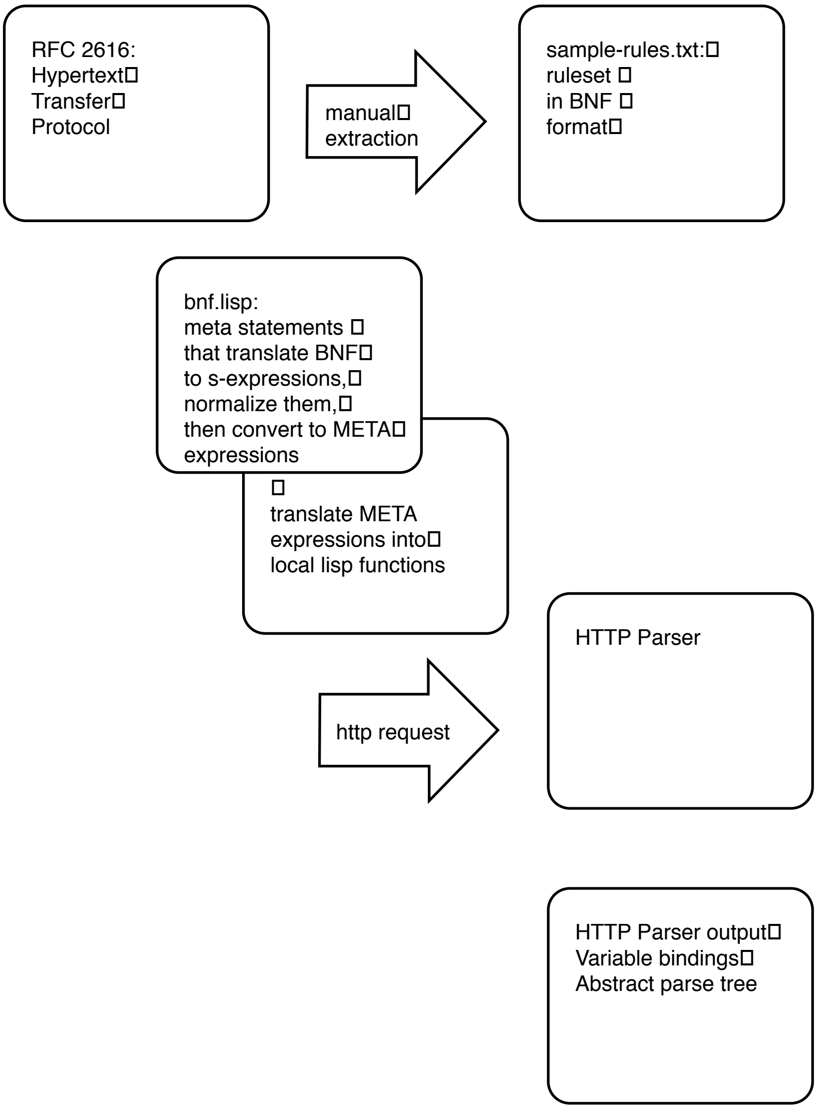
RFC 2616: Hypertext Transfer Protocol
manual extraction
sample-rules.txt: ruleset in BNF format
bnf.lisp: meta statements that translate BNF to s-expressions, normalize them, then convert to META expressions
translate META expressions into local lisp functions
http request
HTTP Parser
HTTP Parser output Variable bindings Abstract parse tree

Figure 3.2: From BNF to a lisp parser

### 3.2.1 Specifying comma-separated lists

BNF allows for a shorthand for specifying comma-separated lists of elements. Separating list elements with commas is not the way of Common Lisp. In the realm of C and Unix, it is quite common. Not surprisingly, I got a definite feeling of going “against the grain” when I was to implement the format for specifying comma-separated lists. In general, the BNF allows for folding linear whitespace (LWS). For comma-separated lists, the picture is more cluttered as it allows for empty elements that do not add to the total count of elements. An expression to match a comma-separated list of two elements should therefore look for something like:

```
*(*lws #\,) *lws element1 *lws #\, *(*lws #\,)
            *lws element2 *(*lws #\,)
```

For most elements, it is appropriate to remove whitespace in front of the element. If this is done elsewhere, we have a simpler regular expression:

```
     *(*lws #\,) element1 *lws #\, *(*lws #\,)
                 element2 *(*lws #\,)
```

As we should consume whitespace in front of a comma, let us write comma as a shorthand for the sequence `*lws \#\,`.

```
        *(comma) element1 comma *(comma)
                 element2 *(comma)
```

If we remove the parenthesis around single elements, and write $1 * \mathit{comma}$ for $\mathit{comma} * \mathit{comma}$, we get:

```
          *comma
                 element1
         1*comma element2
          *comma
```

To generalize the format, we see that the first element will look like *comma element1. Subsequent elements will look like 1*comma element2. After the list we will find *comma.

This format was not quite what I wanted, though. The code ended up as:

```
(defun quantify-expr-list (min max body symbol)
  (declare (ignorable symbol))
  (let* ((comma (list '(kleene-star
                        (kleene-star
                         ((parsed-or-expression #\Return #\Newline)
                          (parsed-or-expression #\Space  #\Tab)))
                        (kleene-star (parsed-or-expression #\Space #\Tab)))
                      #\,))
         (commas (list (cons 'kleene-star comma)))
         (subsequent (list comma commas body)))
    (labels ((nconc-times (n expr)
               (loop for i below n
                     nconc expr)))
      (list commas
            (if (zerop min)
                (list 'parsed-or-expression body 'the-empty-string)
                (cons body
                      (nconc-times (decf min) subsequent)))
            (if (integerp max)
                (nconc-times (- max min 1)
                             (list
                              (list
                               'parsed-or-expression
                               subsequent
                               commas)))
                (list 'kleene-star subsequent))
            commas))))
```

When invoked, the code produced from this would end up in a loop looking for a match on Space or tab, then return or newline, then start looking again. The problem is the line containing

```
(kleene-star (parsed-or-expression #\Space #\Tab)))
```

A kleene-starred expression can match the empty string. If there is a kleene-star around such an expression, we end up with an infinite loop, as the matcher will try to match the empty string as many times as possible. One solution is to remove the kleene-star from the inner expression, and have the outer kleene-star do the repetition.

## 3.3 Match a type rather than characters

The matcher collects characters and builds tokens from these. For each character, a test is done to check whether the character is a token-char or not. So far, the token-chars are specified by a rule that consists of a big "or"-statement.

For a comparison, I add a new rule to the ruleset:

```
          token-type     =  1*token-type-char
```

and I define a corresponding set of characters:

```
HTTP> (deftype rfc2616::|token-type-char| ()
        '(member #\! #\# #\$ #\% #\& #\' #\*
[...]))
RFC2616::|token-type-char|
HTTP> (let* ((string "thisisaverylongtokenendinghere  ")
             (index 0)
             (end (length string)))
        (with-bnf "sample-rules.txt"
                  #.(mapcar (lambda (sym) (intern sym :rfc2616))
                  (list "star-lws" "SP" "HT"
                        "LF" "CR" "segment" "path\_segments"
                        "pchar" "reserved" "escaped" "uric" "unreserved"
                        "alphanum" "lowalpha" "upalpha" "alpha" "digit"
                        "domainlabel" "toplabel" "token-char"))
                  :rfc2616
(values                 (time (dotimes (i 10000)
                                (matchit !(rfc2616::|token|))
                                (setf index 0)))
                        (time (dotimes (i 10000)
                                (matchit !(rfc2616::|token-type|))
                                (setf index 0))))))

;Compiler warnings :
[...]

(DOTIMES (I 10000) (MATCHIT !(RFC2616::|token|)) (SETF INDEX 0))
took 4,186 milliseconds (4.186 seconds) to run.
Of that, 3,460 milliseconds (3.460 seconds) were spent in user mode
         440 milliseconds (0.440 seconds) were spent in system mode
         286 milliseconds (0.286 seconds) were spent
         executing other OS processes.
950 milliseconds (0.950 seconds) was spent in GC.
 29,920,000 bytes of memory allocated.
(DOTIMES (I 10000) (MATCHIT !(RFC2616::|token-type|)) (SETF INDEX 0))
took 318 milliseconds (0.318 seconds) to run.
```

```
Of that, 120 milliseconds (0.120 seconds) were spent in user mode
         70 milliseconds (0.070 seconds) were spent in system mode
         128 milliseconds (0.128 seconds) were spent
         executing other OS processes.
40 milliseconds (0.040 seconds) was spent in GC.
 1,600,000 bytes of memory allocated.
NIL
NIL
HTTP>
```

These forms appear to be good candidates for a speedup. If possible, I want rules translated into deftypes. A rule may be translated to a deftype if it consists of a single-character string, an or-expression of single-character strings, an or-expression of rules that themselves may be translated to deftypes. I could rephrase that. A rule may be translated into a deftype if every element of the rule is either a string of length 1 or an or-expression whose elements are translatable into a deftype.

```
(defun translatable-into-deftype (rhs)
  (typecase rhs
    (string (array-in-bounds-p rhs 0) (not (array-in-bounds-p rhs 1)))
    (list  (and (eq (car rhs) 'bnf::parsed-or-expression)
                (every #'translatable-into-deftype (cdr rhs))))))
```

Next, we want it to actually translate the expression while testing it.

```
(defun translate-to-deftype (rhs)
  (typecase rhs
    (string (when (and (array-in-bounds-p rhs 0)
                       (not (array-in-bounds-p rhs 1)))
              (char rhs 0)))
    (list (and (eq (car rhs) 'bnf::parsed-or-expression)
               (let ((rest (mapcar #'translate-to-deftype (cdr rhs))))
                 (when (every (complement #'null) rest)
                   (cons 'member rest)))))))
```

We still have not handled rules that are composed of disjunct rules, such as

```
alphanum      = alpha | digit
alpha         = lowalpha | upalpha
```

These make for an interesting situation. There are three possible cases for a rule. Either it is not translatable, or it is translatable, or it may be translatable in the future, if we can translate the components into deftypes.

So, when we meet a symbol, we will check whether it is already defined as a deftype. In that case, we can use it as argument in an or-statement.

A fictional rule definition `alpha | ‘‘0’’`, which we see as

`(bnf::parsed-or-expression rfc2616::alpha ‘‘0’’)`

should be converted to a type specification such as `(or alpha (member #\0))`. So, in a parsed-or-expression, we want to group the args according to whether they are strings or symbols. Strings go into a “member” typespec. If there are symbols, we wrap the member typespec in an “or” typespec.

`Deftype` creates a lexical binding. Which is what I want.

Once the deftypes are defined, there remains another hurdle. They are defined in a lexical context, and `BNF::CHAR-OR-TYPE-OR-FUNCTION` does not know about them, because `char-or-type-or-function` is evaluated before these deftypes are evaluated. What to do?

We could hand `char-or-type-or-function` a list of the types that we intend to define.

`with-bnf` calls `meta-rules-and-deftypes`. `meta-rules-and-deftypes` has `deftypes`, and calls `rule-bnf-expr-to-meta`.

We will make `deftypes` into a dynamic variable, so that it's visible from `char-or-type-or-function`.

We can turn off the effect of these locally defined types by commenting out the relevant case in `char-or-type-or-function`, and recompiling `receive-request`.

```
(testcase) ;; with type definitions turned on
(DOTIMES (I 100000) (MATCHIT !(RFC2616::|HTTP-message|)))
took 2,313 milliseconds (2.313 seconds) to run.
Of that, 1,430 milliseconds (1.430 seconds) were spent in user mode
         720 milliseconds (0.720 seconds) were spent in system mode
         163 milliseconds (0.163 seconds) were spent
         executing other OS processes.
842 milliseconds (0.842 seconds) was spent in GC.
 43,259,136 bytes of memory allocated.

(testcase) ;; not using the type definitions
(DOTIMES (I 100000) (MATCHIT !(RFC2616::|HTTP-message|)))
took 3,074 milliseconds (3.074 seconds) to run.
Of that, 2,120 milliseconds (2.120 seconds) were spent in user mode
         590 milliseconds (0.590 seconds) were spent in system mode
         364 milliseconds (0.364 seconds) were spent
         executing other OS processes.
874 milliseconds (0.874 seconds) was spent in GC.
 48,101,792 bytes of memory allocated.
```

Comparing the time spent in user mode, we got a `(- 1 (/ 1430 2120)) = 32.5` percent speedup.

The next improvement I would like to try out is to store indices to the request string rather than strings. Hopefully, this can be done transparently, as we can also modify the accessor methods to fetch the correct subsequence of the string rather than a pair of indices to the start- and endpoint of the relevant string.

```
(push (subseq string past-whitespace index) ,name)))
(push (cons past-whitespace index) ,name)))
```

That made no difference, it seems.

```
(testcase 100000)
(DOTIMES (I N) (MATCHIT !(RFC2616::|HTTP-message|)))
took 2,002 milliseconds (2.002 seconds) to run.
Of that, 1,240 milliseconds (1.240 seconds) were spent in user mode
         530 milliseconds (0.530 seconds) were spent in system mode
         232 milliseconds (0.232 seconds) were spent
         executing other OS processes.
517 milliseconds (0.517 seconds) was spent in GC.
 43,231,024 bytes of memory allocated.
```

Maybe that is because we're replacing one expensive operation with another?

Binding each name to a `(make-array 0 :adjustable t :fill-pointer 0 :element-type 'integer)` and doing:

```
;;  (push (subseq string past-whitespace index) ,name)))
;;       (push (cons past-whitespace index) ,name)))
    (vector-push-extend past-whitespace ,name)
    (vector-push-extend index ,name)))
```

```
(testcase 100000)
(DOTIMES (I N) (MATCHIT !(RFC2616::|HTTP-message|)))
took 2,327 milliseconds (2.327 seconds) to run.
Of that, 1,340 milliseconds (1.340 seconds) were spent in user mode
         820 milliseconds (0.820 seconds) were spent in system mode
         167 milliseconds (0.167 seconds) were spent
```

```
        executing other OS processes.
871 milliseconds (0.871 seconds) was spent in GC.
 43,232,960 bytes of memory allocated.
```

So, this attempt did not generate any difference in speed.

## 3.4 Parse tree

A *parse tree* is an abstract representation of expressions in a language, with a structure similar to that of the production rules of the grammar of the language.

The tree may be represented as statements in the *dot language* of the *Graphviz* package, and look like figure 3.3.

## 3.5 Pragmatic parsing

> To parse a sentence means to recover the constituent structure of the sentence – to discover what sequences of generation rules could have been applied to come up with the sentence
>
> ([Norvig, 1992])

In a way, it implies proving that the sentence is valid in a particular grammar.

Earlier, we touched upon the generation of a parse tree as a result of parsing an HTTP message. At first, the parse tree seemed somewhat cluttered. First, I wanted to chop off the dangling truth values. A Kleene-starred expression has a successful match if it matches nothing. It can return T, but to a human reader, that information just clutters up the view. Removing these left a tree with a better signal to noise ratio.

We rather want to differ between a match that actually matches something, and a match of the empty string, which we would rather forget about. Therefore, we will delete from the parse tree the results of Kleene-star expressions matching the empty string, as well as calls with empty arglist to the concatenating and-operator, cand. That strings are sequences of characters, but they are displayed as a box containing the string, rather than several boxes each containing a letter.

### 3.5.1 Intentional capture with special variables

Index and old-index are made into special variables, so that we don't have to pass them around all the time. This violates referential transparency, that

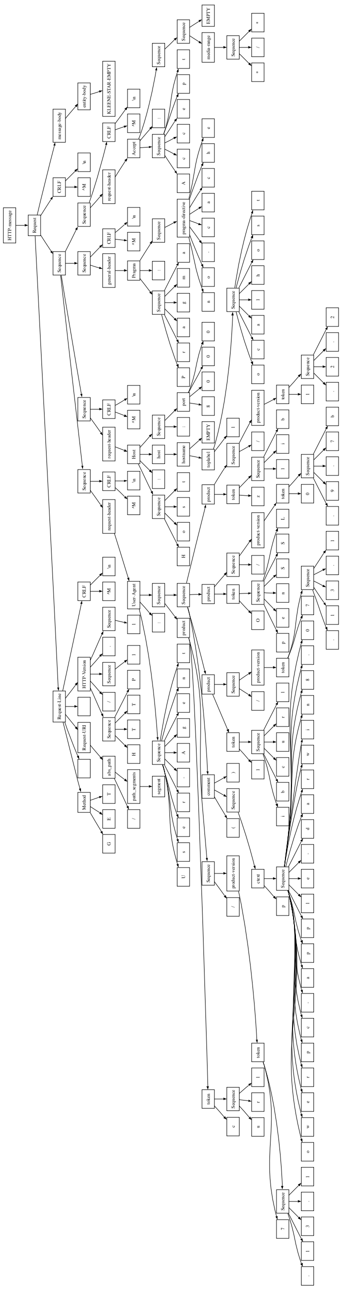

Figure 3.3: Representation of parse tree of an HTTP Request

is, the property that functions may have that says that the function always will have the same output with the same arguments. The gain is that the code is easier to read this way.

### 3.5.2 Keeping the global namespace clean

As the symbol names read from the BNF are interned, we want this done in a separate package to avoid name clashes with names defined in the language or the application.

For example, the ruleset has the following definition:

```
quoted-pair    = "\" CHAR
```

which points back to:

`CHAR = <any US-ASCII character (octets 0 - 127)>`

In Common Lisp, `char` is the name of a function. The big LABELS statement will refer to char, but if it refers to char as part of a rule definition, it should refer to the `rfc2616::char` rather than the `common-lisp:char`. How do we go about that?

If we are further down in the implementation, I may want to refer to `cl:char`, but will I then have to prefix it?

One solution may be to define a package `:rfc2616` for the symbols read from the ruleset, and explicitly prefix any symbol read from the ruleset with `rfc2616:`. Then we will refer to `rfc2616:char` when we mean the char from the ruleset.

When a package is created, it will inherit from `common-lisp-user` by default, so when creating the package for ruleset, we will explicitly ask it not to inherit any symbols, by adding a `:use` keyword with no argument when calling `defpackage`.

### 3.5.3 Using grammar from another RFC

The optional "From:"-header uses grammar from the rfc for internet mail, that is, RFC 822, later superseded by RFC 1123. Ideally, we should be able to use the grammar specified in BNF within these RFCs directly, but there are a few complicating factors.

The BNF notation is a little different, in that it uses the token "/" as an or-token. That is, where RFC 2616 says "A" — "B" to mean: "Either the string "a" or the string "b".", RFC 822 and 1123 say "A" / "B".

For the rules imported from 822 and 1123, I have replaced the characters slash with pipe by hand.

The RFCs have somewhat overlapping and conflicting namespaces, so we have to be careful so as to specify which namespace is the default, and which has to be named explicitly.

One way would be to mechanically check that each of the rulenames in the additional ruleset is unused in rfc 2616. But this would be both cumbersome and prone to human error.

We will instead use a separate namespace for the symbols imported from the mail rfcs. Let's call it `rfc1123`. Currently, the reader imports all symbols and interns them into the `rfc2616` package. We would like a facility to specify another package.

If we wrap the code that calls intern in a let where `*package*` is bound to `:rfc2616`, then unqualified symbols read will be interned in `rfc2616`, but we can still prefix other symbols with a package name.

First step is to use `read-from-string` rather than intern.

```
CL-USER> (intern "foo" :common-lisp-user)
|foo|
NIL

CL-USER> (let ((*package* (find-package :common-lisp-user)))
           (read-from-string "foo"))
FOO
3
```

Note that intern preserves the case of the symbol. Read can preserve it, too.

```
CL-USER> (let ((*package* (find-package :common-lisp-user))
               (*readtable* (copy-readtable nil)))
           (setf (readtable-case *readtable*) :preserve)
           (read-from-string "foo"))
|foo|
3
```

Then we define an empty package for the imported symbols by adding the `(:use)` flag to `defpackage`.

```
(defpackage #:rfc1123 (:use))
#<Package "RFC1123">

(let ((*package* (find-package :common-lisp-user))
               (*readtable* (copy-readtable nil)))
           (setf (readtable-case *readtable*) :preserve)
           (read-from-string "rfc1123::foo"))
```

```
; Evaluation aborted
; as the package rfc1123 is not known.

(let ((*package* (find-package :common-lisp-user))
               (*readtable* (copy-readtable nil)))
           (setf (readtable-case *readtable*) :preserve)
           (read-from-string "RFC1123::foo"))
RFC1123::|foo|
12

; with an upcased designator for the package, it is found

(let ((*package* (find-package :common-lisp-user))
               (*readtable* (copy-readtable nil)))
           (setf (readtable-case *readtable*) :upcase)
           (read-from-string "rfc1123::foo"))
RFC1123::FOO
12
```

`Parse-rule-definition` reads through the string and parses the definitions.

Instead of `(setf name (intern (subseq string old-index index) package))))` we will use `(setf name (let ((*package* package)) (read-from-string (subseq string old-index index))))` As a first step, we would like to include the grammar from `rfc1123` with prefixed names. Later, we may want to put them in a separate file, and associate this file with a particular package name.

Well, as : is not allowed as part of a token, we have to alter our reader a little.

```
;; (with-match [@token-char \$@token-char]
;;    (setf thing (intern (subseq string old-index index) package))))
   (with-match [@token-char \$@token-char
                             {[\#\\: \#\\: @token-char \$@token-char][]}]
     (let ((*package* package))
       (setf thing (read-from-string (subseq string old-index index))))))
```

With this alteration, the external names are interned in their respective packages:

```
(pprint (bnf::parse-rule-definition
"         rfc1123::mailbox        = rfc822::addr-spec ; simple address
                                    | [rfc822::phrase]
```

```
                                        rfc822::route-addr; name & addr-spec"
0 :package (find-package :rfc2616)))

(BNF::RULENAME RFC1123::MAILBOX
 ((RFC822::ADDR-SPEC) (BNF::OR-TOKEN)
  ((BNF::PARSED-OR-EXPRESSION ((RFC822::PHRASE)) BNF::THE-EMPTY-STRING))
  (RFC822::ROUTE-ADDR)))
; No value
```

With all the names prefixed with a package name, the expressions are rather unreadable. Another option would be to store the expressions in a separate file, associate a package with that file, and intern all unqualified names from that file in the specified package.

This way, the local function `RFC2616::FROM` refers to `RFC1123::MAILBOX`, which in turn refers to `RFC822::ADDR-SPEC` et cetera.

### 3.5.4 Specifying the format of the MIME file in BNF

Now that I have a system for generating parsers from BNF, I wanted to teach the system how to interpret the format of the Multipurpose Internet Mail Extensions (MIME) file. From the top of the file:

```
# This is a comment. I love comments.

# [... More comments ...]

# MIME type                      Extensions
application/activemessage
application/andrew-inset         ez
application/applefile
application/atomicmail
```

We are interested in the lines such as “application/andrew-inset ez”. This line tells that files whose name ends in “.ez” have the mime type “application/andrew-inset”.

As an application of the *with-bnf* framework, we could write a file `mime-filter.bnf`, containing a bnf definition of the file format:

```
;;;# mime-filter
mime-list = (comment  | type-suffixes | type | blank) newline
newline = "
"
comment = "#" *NOT-NEWLINE
```

```
blank = *" "
type-suffixes = type (" "|" ") *(" "|" ") suffixes
type = token "/" token
suffixes = suffix *suffix
suffix = token
token = TOKEN-CHAR *TOKEN-CHAR
```

and collect lines from the mime-type file like this:

```
(defparameter *mime-suffix-alist*
  (let* ((string (file-to-string "/private/etc/httpd/mime.types"))
         (index 0)
         (end (length string))
         (collection ()))
    (with-bnf "mime-filter.bnf"
      '(#\Space) :mime
      (loop
          (unless (matchit !(mime::|mime-list|)) (return collection))
          (when (shiftf mime::|type-suffixes| nil)
            (apush (shiftf mime::|suffix| nil) mime::|type| collection))
        (setf mime::|type| nil))))
  "An association list where the first element is a list of suffix strings
such as (\"jpeg\" \"jpg\"), and the second element is a mime designator
such as \"image/jpeg\".")
```

In addition, we need to lookup the mime-type of a particular suffix.

## 3.6 Checking for type definitions

The function `char-or-type-or-function` takes a name as its argument, and returns a string in the META language. There are built-in predicates for most of the available cases, but for checking whether there is a type whose name is the name, is harder. I have used OpenMCLs built-in predicate `type-specifier-p` to check whether there exists a matching type, which at least in practice works as desired.

## 3.7 Ignoring whitespace

I've got a problem matching a header that looks like this, as there is whitespace occuring between the the header name and the header value:

`Host:  127.0.0.1:8000`

This should match the rule that is on the form:

```
Host           = "Host" ":" host [ ":" port ]
```

The challenge is that the spec also says the following:

> implied *LWS The grammar described by this specification is word-based. Except where noted otherwise, linear white space (LWS) can be included between any two adjacent words (token or quoted-string), and between adjacent words and separators, without changing the interpretation of a field. At least one delimiter (LWS and/or separators) MUST exist between any two tokens (for the definition of "token" below), since they would otherwise be interpreted as a single token.

I will therefore need to skip whitespace, but in some cases, the whitespace may be part of the rule. Therefore, if the match fails, I will also need to try to match with the whitespace. I will try with the whitespace included after failing without the whitespace, as I presume that there will be more rules that discard the whitespace than those who use it.

Whitespace is informally specified, and as there is no separate scanning procedure, we deal with it as we read. There are just a few rules that want to se the prepended whitespace, and we'll remember those separately. Unless a rule requires whitespace, I will add an expression in front of the rule that munges whitespace.

## 3.8 From a flat parse to a parse tree

The hostname of the server can be supplied both as part of the first line of the request, if the URL is a full URL called an `AbsoluteURI`, or it can be specified as the request-header "Host:". The RFC specifies that the `AbsoluteURI` has precedence over the request-header. Therefore we need to know from where in the request the name of the host origins.

Baker's paper envisions a flat parse, that is, one where the result of the indivudual matches is bound to a symbol, or stored through side effects. I would prefer the result of the parse to be hierarchical, in which we would get not only the values of the tokens, but their structure as well.

Currently, the `with-bnf` macro generates code whose return value is t if there was a match, nil if there was none. The value is pushed onto the list associated with the name of the rule.

Instead, we would like matchit to return a tree, in which a node has a name, a sequence of characters, and points to its child nodes.

At the level of match, it is ok not to return a useful value. So I would like the calls to functions to return useful values. If we are to match a concatenation, we'll have to collect these, and store the parts as a list.

As of now, a form such as

```
(MATCHIT [!(RFC2616::|Method|)
          @RFC2616::SP
          !(RFC2616::|Request-URI|)
          @RFC2616::SP
          !(RFC2616::|HTTP-Version|)
          !(RFC2616::CRLF)])
```

expands to:

```
(LET ((META::OLD-INDEX INDEX))
  (OR (AND (RFC2616::|Method|)
           (META::MATCH-TYPE RFC2616::SP)
           (RFC2616::|Request-URI|)
           (META::MATCH-TYPE RFC2616::SP)
           (RFC2616::|HTTP-Version|)
          (RFC2616::CRLF))
      (NOT (SETF INDEX META::OLD-INDEX))))
```

but in order to generate a parse tree, AND should be a collector as well as a truth test.

Then we will need a test cand that not only tests whether all arguments are true, but also collects the results of the tests, and returns them.

A few of the expansions in `meta::compileit` (concatenation and kleene-star) had to collect their results, and the generic character and string matchers had to collect the tokens they had matched.

Now, let us see whether we can use this information. Given a parse tree, we start by locating the paths to values for host, as a sort of epistemological trail for the value. Given a parse treee, we ask the function opionion-of with the arguments whom, regarding, and where, and with the answers, we can adhere to the semantics specified in chapter 5.2. A test shows them in practice:

```
      chr@macchr/~/proj/privat/jobhunt\$ telnet localhost 8000
Trying ::1^M
telnet: connect to address ::1: Connection refused
Trying 127.0.0.1^M
Connected to localhost.
```

```
Escape character is '^]'.
HEAD / HTTP/1.1
Host: localhost
Connection: Keep-Alive

^M
HTTP/1.1 200 OK^M
Date: Fri, 27 May 2005 07:11:01 GMT^M
Server: Clack/0.1 (Darwin)^M
Connection: Keep-Alive^M
Content-Length: 5240^M
Content-Type: text/html^M

# the server uses the Host: header if nothing else is said

^M
HEAD http://localhost/ HTTP/1.1
Host: macchr
Connection: Keep-Alive

^M
HTTP/1.1 200 OK^M
Date: Fri, 27 May 2005 07:11:27 GMT^M
Server: Clack/0.1 (Darwin)^M
Connection: Keep-Alive^M
Content-Length: 5240^M
Content-Type: text/html^M

# the host part of the complete url has precedence over a conflicting
# Host:-header

^M
HEAD / HTTP/1.1
Host: macchr
Connection: Keep-Alive

^M
HTTP/1.1 200 OK^M
Date: Fri, 27 May 2005 07:11:42 GMT^M
Server: Clack/0.1 (Darwin)^M
Connection: Keep-Alive^M
Content-Length: 104^M
Content-Type: text/html^M
```

```
^M

# a different page, of length 104, is served for a virtual host
```

## 3.9 Memoization

To store the result of an expensive function call, and return this result if the function is called again with the same arguments is called memoization. As macroexpanding `with-bnf` took a long time, I have added memoization code to its definition.

# Chapter 4

# The response

Once the request has been prosessed, we will want to find a resource that matches what the client asked for, thereafter we may look for a variant of that resource, and finally we will serve it as an entity.

## 4.1 The resource

Resources are identified by a URI, but a resource may be available in multiple representations.

## 4.2 The variant

A variant is a particular representation of a resource.

## 4.3 The entity

The entity is the payload of a request or reponse. It encompasses both the entity-headers and the optional entity-body.

## 4.4 Semantics of HTTP

From an implementor's point of view, it seems odd to me that the productivity enhancement enabled by the inclusion of BNF is not also utilized when it comes to the *semantics* of HTTP. As internet standards are implemented in several programming languages, several operating systems and several architectures, it should make sense to centralize some of the workload of implementing a standard by providing machine-readable pseudocode that outlines the intended behaviour of an application. As there is no such representation, only the commitee-authored prose describing the rules of the dialogue between clients, caches and origin servers, we'll resort to writing fairly readable code that can then be macroexpanded into working code. One approach is to build a language to talk about the problem domain, in this case the HTTP protocol and its use, and in parallell, extend the language with a substrate level to reach these abstractions [Graham, 1994].

It may be useful to write tests that verify that the server has a particular trait specified by the RFC.

Rainer Joswig has written an entry on dealing with so-called Domain-Specific Languages (DSLs) in Lisp [Joswig, 2005]. He calls his development style *"Putting parentheses around the specification and make it run."* In short, it's about molding the written specification of a DSL into working code through using the interactive features of lisp.

My take on this would be to highlight the parts of the specification that deal with the semantics of an origin server, that is, take out what is about syntax, clients, and proxies and caching, and write code that implements what is left.

### 4.4.1 A language for semantics?

Backus-Naur form is quite useful when implementing a parser for a foreign format. When it comes to the semantics of HTTP, we do not have such a language. In the specification of MD5 [Rivest, 1992], which I followed for my own implementation, there is a sample implementation written in naïve C code, which is very useful for the implementor, as unintended "room for doubt" is resolved in the code. In the Common Lisp Hyperspec [Guy L. Steele, 1990], section 2.2 The Reader Algorithm, there is an example of a specification of an algorithm for reading a character in Common Lisp. This is also a useful approach to giving implementors a hand in following the specification and ending up with a conforming implementation.

## 4.5 Sending the response

Both requests and responses consist of an entity. A resource is something that is pointed to by a URI. The entity sent as a response is in most cases a manifestation of a variant of a resource.

A server has virtual hosts. Virtual hosts have resources. Resources respond to methods. As resources may either point to functions or files, we may say that these occupy the same namespace. We choose to let functions have precedence over files. A function should answer to a method. If the function does not handle a method, it may reply that it does not.

A resource should know which methods it supports. Methods can be safe, that is, they do not have side-effects. They can be idempotent, in that several requests have the same effect as a single request.

Say that each resource has an alist of METHOD-to-response, wherein response is a function that takes the request as argument, and returns a response. We will have a default set of such answers.

Currently, the function answer-request replies to all requests. Per default, it replies as if to a `GET`; but if the method is not a `GET`, it will not print a body to the output stream. This is a quite simple solution, a better one would be to handle `OPTIONS`, `HEAD`, and conditional `GET`, and reply “method not supported” to other methods.

There is also an interesting distinction between using `#'foo` and `'foo`. With `#'foo` you get the function cell of the symbol, and thus you loose if you later want to alter that function.

## 4.6 Speeding up the file transfer

Some instances of web servers spend most of their busy time serving static files. I made a comparison between the apache httpd server and my own server. My server was not as fast as the Apache httpd, so I looked into what could be the reason. On operating systems that support it, the apache server will use the sendfile system call, which asks the operating system to send a file from disk to a network socket in a single system call. *Sendfile* is available in linux, and Windows NT has a similar call, named *transmitfile*. My host OS, Mac OS X, does not yet provide such a system call.

The fallback for the apache httpd is to call *writev*, which writes from multiple buffers in a single system call. I wanted to try this from Common Lisp. OpenMCL, the implementation of Common Lisp that I use, has an interface to C system calls.

I did a comparison between my server and the Apache httpd on serving a 2.5 megabyte file. Apache was consistently twice as fast. I then rewrote my code, and got a speedup.

I asked on the mailing list for the openmcl lisp implementation, and got a tip on using the actual file descriptors, as they are seen from C.

First, how apache performs on a 20 megabyte file on the loopback interface, with 10 repeated downloads. I collected the reported MB/s speed from wget, and looked at the average.

```
(avg 7.50 6.56 12.69 6.75 7.01 10.27 6.94 8.27 14.74 7.75) 8.84
```

my own code:

`(avg 5.45 10.78 5.73 4.90 1.44 2.82 6.05 8.00 5.88 10.22) 6.12`

Next, how the code suggested by Gary Byers performs:

`(avg 6.06 7.23 8.68 7.31 6.92 9.92 7.86 5.95 11.03 7.19) 7.81`

There is a nice speedup when using file descriptors, but I can't yet get the same performance as the apache server.

On asking what to do when write returned `-35 (Socket is temporarily unavailable)`, I got two suggestions from Gary Byers, one is to use the `select(2)` call, and thus let the thread wait until the socket is ready for writing, another is to use openmcls built-in macro `with-eagain`, which will write until the response code is positive.

## 4.7 MD5

MD5 [Rivest, 1992] is a means to ensure that a message has not been altered since the checksum was computed. Erik Naggum [Naggum, 2005] claims that the specification of MD5 indicates a procedure that is to be implemented close to the hardware, that is, in assembly language or C. Such an implementation is beyond my scope, and I have instead implemented the algorithm in Common Lisp, aiming for a terse rather than fast implementation.

In HTTP, MD5 is used as a checksum for the entity, and also for *Digest Authentication*, as specified in RFC 1321 [Franks et al., 1999]. It works by setting up four registers, a, b, c, and d, and a set of four operations. 16 32-bit words of the message are used in turn, and for each turn, these 32 bits are used in calls to the operations, which affect the contents of the four registers. When the message is exhausted, the four registers are used to produce the string that is the MD5 checksum.

I have little reason to doubt the advice on writing this part in assembler. For example, there are assembler instructions that will rotate the bits in unsigned an 32-bit integer with a single call, while my code that does the same expands to about 45 lines of assembly code.

# Chapter 5

# The server

Figure 5.1 displays the dialogue between the browser and the server, in terms of the TCP model.

The TCP/IP stack model stems from the book Unix Network Programming [Stevens et al., 2003]. Server and browser both communicate through HTTP.

In the most generic version, a server replies to a single request, then closes the connection. We would like it to be able to keep on answering requests, and servicing each client until it wants to close the connection.

Currently there is `open-server`, which opens a listener for connections on port `8000`. `run-server` gets handed the listener, and each time there is a response from `accept-connection`, a new process is created running `accept-and-receive`.

`accept-and-receive` gets the connection as a stream, and as long as `rfc2616:receive-request` returns `t`, it will be called. When it returns `nil`, the stream is closed, and the process dies.

## 5.1 Multiple processes

The openmcl function `process-run-function` runs a process as a native thread.

> For a variety of reasons - better utilization of CPU resources on single and multiprocessor systems and better integration with the OS in general - threads in OpenMCL 0.14 and later are *preemptively scheduled*. In this model, lisp threads are native threads and all scheduling decisions involving them are made by the OS kernel.

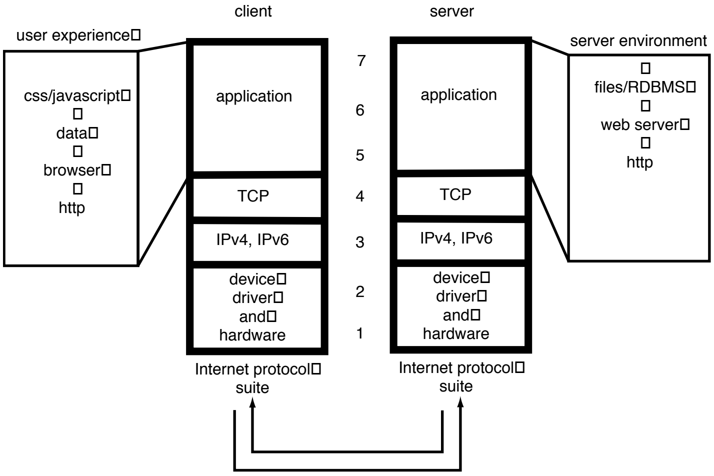

Figure 5.1: Web server and client in the context of TCP stacks

## 5.2 Signal handling

A unix process should be able to handle received signals.

## 5.3 Error handling

If there is a socket error, receive-request will return nil so that answer-request will close the socket . This is not too sophisticated, but ensures that a socket is closed if the client shuts down the connection.

## 5.4 Profiling

When an application is running, it may be possible to ask it what it is doing. Profiling is collecting many such samples. The functions where the application spends most of its time, may be bottlenecks. Once we know which functions to look at, we can either improve on the overall algorithm of the application in order to call particular functions less, or optimize the functions where the application spends most of its time, ideally through the algorithm or through typing and the like.

At the time of writing, OpenMCL does not have a built-in profiler, but there is such a facility in the development environment called SLIME [Eller, 2005]. Profiling is activated per-package, like this:

```
CL-USER>  (mapcar (lambda (package)
                (swank:profile-package (find-package package) t t))
```

```
              (list :http :server :utilities :bnf :meta :common-lisp))

;;; use the application

CL-USER> (mon:report)
Function                      Time    Cons  Calls  Sec/Call   Call   Time     Cons
--------------------------------------------------------------------------------
RFC2616:RECEIVE-REQUEST:     98.41  289.39     52  0.100029  68406  5.202  3557136
SERVER::ACCEPT-CONNECTION:  16.72    9.96      4  0.220991  30616  0.884   122464
BNF::MAKE-RULEMATCH:          6.91    0.00   7520  0.000049      0  0.365        0
UTILITIES::DELETE-IN-TREE:   4.36    0.00   7591  0.000030      0  0.231        0
RFC2616::READ-TO-CRLFCRLF:   2.92    0.00     52  0.002972      0  0.155        0
RFC2616::HTTP-LOG:            2.28   29.84     52  0.002318   7053  0.121   366776
--------------------------------------------------------------------------------
TOTAL:                      131.62  329.20  15271                    6.957  4046376
Estimated monitoring overhead:  0.14 seconds
Estimated total monitoring overhead:  0.52 seconds
One hundred eighteen monitored functions were not called.
See the variable mon::*no-calls* for a list.
; No value
```

## 5.5 Dumping the application

In Common Lisp, *"dumping"* means to save the state of the lisp image. The openmcl function `save-application` saves the state of the current lisp. One of the keyword arguments is `init-file`, I gave it the filename of an init-file, `clack-init.lisp`, and saved a file with that name in my home directory. The file contains the line `(server:start)`.

I also had to specify that I wanted the lisp kernel in the image. Therefore I asked for ``` ``:prepend-kernel t'' ```.

Thus I can start the server by double-clicking it's icon. This launches a terminal with a lisp listener. As a starting point, that is sufficient.

`Save-application` seems tailored to window applications, as it has a keyword for a toplevel-function and this function will be called repeatedly when the application is launched. I guess toplevel-function should be a loop that checks for particular events. For the server, which has it's own loop, it is not so useful.

# Chapter 6

# Further work

I now have a working web server. The next steps would be to fully support the various methods and combinations of headers in order to reach `HTTP 1.1` compliance. Thereafter, I would like the server to handle sessions, so that clients that return their cookies hace session objects realted to it, and those who do not return the cookies have a fallback scheme, such as URL rewriting.

With a well-rounded web server, the next steps would be to add elements of a what is informally called a *web server stack*. That is the web server with an operating system, a database server and the server itself, perhaps with a load-balancing arrangement.

## 6.1 Methods and combinations

### 6.1.1 Methods

At this juncture, the server replies to `GET` and `HEAD` requests. For serving plain files, the methods `POST`, `PUT`, `DELETE` and `CONNECT` are all fairly irrelevant, so I would likke to answer "not implemented" to all of these. `OPTIONS` should return a list of `GET`, `HEAD` and `OPTIONS`. In addition to the HTTP methods, there are various combinations of headers that require special attention.

### 6.1.2 Conditional requests

The server handles `If-Modified-Since`, but there are other variants on this theme.

`If-Unmodified-Since` is relevant to a request that will alter the resource, such as `PUT` or `POST`. The same goes for `If-Match`, but rather than comparing time, this compares an entity tag, a string that should identify a particular version of a resource.

`If-None-Match` has about the same semantics as `If-Modified-Since`.

`If-Range` has either a date or a tag as its value, and implies that the range specified by the Range-header should be sent if the entity matches, or resource has not been modified since the date.

### 6.1.3 Ranges

Alhthough it is nice to have, it is not mandatory to support byteranges. The web server may well reply as if the range header field had not been there.

### 6.1.4 Content negotiation

Given a particular resource, a web server may attempt to tailor the response to the preferences of the client.

**Media-type** If the resource is available in several formats, or an automatic translation is possible, the the server should choose the appropriate format.

**Character sets** For some character sets, it may be possible to make a translation from one to the other, if the source character set is `ISO-8859-1`, then it is easy to translate it to *Unicode* and encode it as `UTF-8`. A translation from Unicode to a character set in the ISO-8859-set may be possible.

**Encoding** Encoding is mostly about compression. If a compression scheme is available and the client wants it, the server should not only compress the reply, but also attempt to cache the compressed version of the resource in order to be able to reply even faster to such a request at a later time.

**Language** Truly advanced applications may have support for several human languages, and of course, if there is a match with one of the preferred languages of the client, it is adviseable to adhere to it.

## 6.2 Session handling

For some web applications, it is useful to attach a session object to a set of requests from the same client. The proverbial example is the shopping basket of an online store. The basket will be attached to a session object, and the client will be given a unique string and it is expected that the client enclose this string with each request to the same server.

### 6.2.1 Cookies

The regular way of sending the ticket is through the cookie mechanism. Cookies are named after the chinese cookies that restaurant guests get after their meal. Inside the cookie is a piece of paper with some word of wisdom. In HTTP, the cookie is invisible to the user, and its value is connected to the name of the server. The value is sent back only when the client accesses the same server. The cookie has a timeout value. Although data may be stored in the cookie, it is usually a key to a database on the server side, rather than information in itself.

### 6.2.2 URL rewriting

To detect whether the client supports cookies, the server will send a cookie with the firsts reply, and if that cookie is returned, the server can assume that the client will continue to send them. If the cookie is not returned and it session support is required for the web application to work, the client should either be alerted that cookie support is mandatory, or the server may implement session support through rewriting the URLs.

In URL rewriting, a unique string is generated and hidden within the location of the response. As long as links within the response are relative, or rewritten in the same manner, the server may acquire a unique id for a session object from each request.

## 6.3 Web server stack

Certain combinations of operating system, web server, database server and server-side scripting languages are known to work well together. Microsoft products are often deployed together, and then the stack consissts of Windows NT as the operating system, Internet Information Server as the web server, SQL server as the database, and C# as the scripting language. The predominant open-source stack consists of Linux or a BSD variant as the operating system, Apache as the web server, MySQL or PostgreSQL as the database, and Perl or PHP as the server-side scripting language.

### 6.3.1 Operating system

It is expected that the operating system provide an abstraction of the network layer in the form of a TCP/IP and sockets implementation.

### 6.3.2 Database server

**RDBMS** A dynamic web server is perceived as one whose contents are expected to change frequently and provides ways of altering the contents. Most dynamic web servers rely on a database for storage. The database is mostly a Relational Database Management System (RDBMS). In order to minimize the overhead of connecting and disconnecting to the databse, the webserver may have a continuous connection to the database server. Pages will mostly be constructed through requesting a set of rows from the database, and loop over these rows, generating html code as a means of presentation.

Applications in which the presentation, business logic and database abstraction are separated, are called three-tiered applications.

**Persistent storage** An alternative to the traditional approach of storing records in the database and querying the database when they are to be displayed, is to store everything in memory, so called Object Prevalence [Caekenberghe, 2003]. When something in memory is to be altered, the change is logged to a file. Once a while the image itself may be dumped to disk as a backup. Should the state of the web server be lost because of a reboot or something similar, the web server will restart with the previously saved image, apply the log as a set of patches, and thus restore the state to what it was before the reboot. The upside of this approach is that it is adapted to the regular usage pattern of a web server, in that the information is seldom updated, and that information is mostly read, rather than updated. The downside is that this is an unusual way of developing web applications, and in addition, there's not much of a use for a database server, as this may well be done through a lisp datastructure as long as the alterations to the structure is logged as a set of expressions.

### 6.3.3 Server-side scripting language

At first there was a division between static pages, and pages generated by Common Gateway Interface (CGI) scripts, that is, a technique in which the web server would initiate a unix environment with the values of the HTTP request bound to variables, call the script within this environment, collect the output from the script, and return it to the client. Then came server-side scripting languages, that is, languages that were run on the server side, but not as a CGI, but rather in the server proper. The first implementations were Netscapes `NSAPI` and soon after, Microsofts `ISAPI`. These made it possible to run code written in JavaScript or VBScript on the server side without the overhead associated with the incvocation of a CGI script. Later on came PHP: Hypertext Preprocessor (PHP), employing a language dedicated to generating web pages on the server side. When running, these languages all have access to a server object, an application object, a session object and a request object. With a server written in Common Lisp, it would be natural to employ an arrangement for mixing html and server-side programming such as Lisp Server Pages (LSP) [Wiseman, 2002].

### 6.3.4 Authentication

These days, http-based authentication is barely employed, so although implementing it would be a nice exercise, it will probably not be used, as server administrators and programmers prefer to cook their own username-password pageson top of the session support.

### 6.3.5 Secure Sockets support

Secure Sockets Layer (SSL), on the other hand, is neccessary for most commercial endeavours on the web. These solutions require a somewhat expensive sertificate from a one of several named authorities, as well as either interfacing with C libraries or writing an implementation in the native language of the rest of the web server.

### 6.3.6 Load balancing

If the load of a web server is to be shared among several machines, so-called load balancing is employed. This may be done through redirecting requests to one of many servers, by having a main server acting as a proxy for other servers behind it, or through dns juggling wherein dns servers will alter their answer for the web server.

# Chapter 7

# Conclusion

Parsing a foreign syntax is easier with BNF. The messy part is the semantics of the protocol. We have come to the point where the server needs a big “CGI”-application, one that answers requests properly. By using BNF expressions, the task of writing a parser is made a lot faster. This gain in productivity should also be available for implementing the semantics part of a server.

The semantics could be specified using prose, as in the specification of the Common Lisp reader, as naïve computer code as done in the specification of MD5, or in a dedicated language. The open-ended question is what this language should be like.

# Bibliography

[Andreesen, 1993] Andreesen, M. (1993). proposed new tag: IMG. `http://www.webhistory.org/www.lists/www-talk.1993q1/0182.html`.

[Anklesaria et al., 1993] Anklesaria, F., McCahill, M., Lindner, P., Johnson, D., Torrey, D., and Albert, B. (1993). The Internet Gopher Protocol (a distributed document search and retrieval protocol). RFC 1436 (Informational).

[Baker, 1991] Baker, H. G. (1991). Pragmatic parsing in Common Lisp. *ACM Lisp Pointers*, 4(2):3–15. `http://home.pipeline.com/~hbaker1/Prag-Parse.html`.

[Barlow, 2004] Barlow, D. (2004). Asdf - another system definition facility. `http://constantly.at/lisp/asdf/index.html#Top`.

[Bradner, 1996] Bradner, S. (1996). The Internet Standards Process – Revision 3. RFC 2026 (Best Current Practice). Updated by RFCs 3667, 3668, 3932, 3979, 3978.

[Byers, 2005] Byers, G. (2005). OpenMCL. `http://openmcl.clozure.com/`.

[Caekenberghe, 2003] Caekenberghe, S. V. (2003). Rebel with a cause. `http://homepage.mac.com/svc/RebelWithACause/#prevalence`.

[CERN, 1996] CERN (1996). Status of the cern httpd. `http://www.w3.org/Daemon/`.

[Eller, 2005] Eller, H. (2005). *The Superior Lisp Interaction Mode for Emacs.* `http://common-lisp.net/project/slime/doc/html/slime.html`.

[Fielding et al., 1997] Fielding, R., Gettys, J., Mogul, J., Frystyk, H., and Berners-Lee, T. (1997). Hypertext Transfer Protocol – HTTP/1.1. RFC 2068 (Proposed Standard). Obsoleted by RFC 2616.

[Fielding et al., 1999] Fielding, R., Gettys, J., Mogul, J., Frystyk, H., Masinter, L., Leach, P., and Berners-Lee, T. (1999). Hypertext Transfer Protocol – HTTP/1.1. RFC 2616 (Draft Standard). Updated by RFC 2817.

[Fielding et al., 2003] Fielding, R., Gettys, J., Mogul, J. C., Frystyk, H., Masinter, L., Leach, P., and Berners-Lee, T. (2003). Hypertext Transfer Protocol – HTTP/1.1. Internet RFC.

[Fisher, 2003] Fisher, D. (2003). Link Prefetching FAQ. `http://www.mozilla.org/projects/netlib/Link_Prefetching_FAQ.html`.

[Foderaro, 1999] Foderaro, J. (1999). AllegroServe - a Web Application Server. Web site. `http://opensource.franz.com/aserve/index.html`.

[Franks et al., 1999] Franks, J., Hallam-Baker, P., Hostetler, J., Lawrence, S., Leach, P., Luotonen, A., and Stewart, L. (1999). HTTP Authentication: Basic and Digest Access Authentication. RFC 2617 (Draft Standard).

[Gabriel, 1990] Gabriel, R. P. (1990). Good News, Bad News, How to Win Big. `http://www.ai.mit.edu/docs/articles/good-news/good-news.html`.

[Garshol, 1998] Garshol, L. M. (1998). BNF and EBNF. `http://www.garshol.priv.no/download/text/bnf.html`.

[Gillies and Cailliau, 2000] Gillies, J. and Cailliau, R. (2000). *How the web was born*. Oxford University Press.

[Graham, 1994] Graham, P. (1994). *On Lisp*, chapter 1, pages 3–5. Prentice-Hall.

[Guy L. Steele, 1990] Guy L. Steele, J. (1990). *Common Lisp The Language*. Digital Press, second edition.

[Joswig, 2005] Joswig, R. (2005). Dsls in lisp, an example for an advanced dsl development technique in lisp. `http://lispm.dyndns.org/news?ID=NEWS-2005-07-08-1`.

[Kantrowitz, 1991] Kantrowitz, M. (1991). Defsystem. `http://www-cgi.cs.cmu.edu/afs/cs/project/ai-repository/ai/lang/lisp/code/tools/defsys/0.html`.

[Mallery, 1994] Mallery, J. C. (1994). A common lisp hypermedia server. WWW-94 Conference Paper. `http://www.cl-http.org:8001/cl-http/ai/projects/iiip/doc/cl-http/server-abstract.html`.

[McIntyre, 2005] McIntyre, R. (2005). Mr. Moore in the Datacenter. `http://www.ryanmcintyre.com/2005/03/mr_moore_in_the.html`.

[Naggum, 2005] Naggum, E. (2005). lognot på 32-bits integer. Usenet message. `http://groups.google.com/group/no.it.programmering.lisp/msg/0842bbe5e3dd1671?dmode=print`.

[NCSA HTTPd Development Team, 1997] NCSA HTTPd Development Team (1997). Ncsa httpd overview. `http://hoohoo.ncsa.uiuc.edu/docs/Overview.html`.

[Nielsen, 1997] Nielsen, J. (1997). The need for speed. `http://www.useit.com/alertbox/9703a.html`.

[Norvig, 1992] Norvig, P. (1992). *Paradigms of Artificial Intelligence Programming: Case Studies in Common Lisp*, chapter 19, page 656. Morgan Kaufmann Publishers, Inc.

[Rivest, 1992] Rivest, R. (1992). The MD5 Message-Digest Algorithm . RFC 1321 (Informational).

[Sarrocco, 2002] Sarrocco, C. (2002). Improving ip connectivity in the least developed countries. `http://www.itu.int/osg/spu/ni/ipdc/study/Improving%20IP%20Connectivity%20in%20the%20Least%20Developed%20Countries1.pdf`.

[Schmidt, ] Schmidt, J. Portable allegroserve. `http://portableaserve.sourceforge.net/`.

[Stevens et al., 2003] Stevens, W. R., Fenner, B., and Rudoff, A. M. (2003). *UNIX Network Programming: The Sockets Networking API*, volume 1, chapter 1.7. Addison-Wesley, 3 edition. `http://proquest.safaribooksonline.com/?x=1&mode=section&sortKey=title&sortOrder=asc&view=&k=null&g=&catid=itbooks.network.unix&s=1&b=1&f=1&t=1&c=1&u=1&r=&o=1&n=1&d=1&p=1&a=0&xmlid=0-13-141155-1/ch01lev1sec7`.

[The Apache Group, 2005] The Apache Group (2005). Welcome! - the apache http server project. `http://httpd.apache.org/`.

[Wiseman, 2002] Wiseman, J. (2002). Lisp server pages. `http://lemonodor.com/archives/000128.html`.

# List of Figures

3.1 The balance between decidability and expressiveness . . . . . 21
3.2 From BNF to a lisp parser . . . . . . . . . . . . . . . . . . . . . . 24
3.3 Representation of parse tree of an HTTP Request . . . . . . . 32

5.1 Web server and client in the context of TCP stacks . . . . . . 47

C.1 Functions in the utilities package. . . . . . . . . . . . . . . . . . . 63
C.2 Functions in the meta package. . . . . . . . . . . . . . . . . . . . 63
C.3 Functions in the bnf package. . . . . . . . . . . . . . . . . . . . . 64
C.4 Functions in the http package. . . . . . . . . . . . . . . . . . . . . 65
C.5 Functions in the server package. . . . . . . . . . . . . . . . . . . . 66

# Appendix A

# Colophon

For the programming, I used openmcl, an implementation of Common Lisp for PowerPCs running Mac OS X, Darwin or LinuxPPC. I ran it on an Apple Powerbook 1.33ghz running Mac OS X 10.4.2. OpenMCL interfaced with XEmacs 21.4 through SLIME.

# Appendix B

# Acronyms

**HTTP** Hypertext Transfer Protocol

**RFC** Request for Comments

**BNF** Backus-Naur form

**XML** Extensible markup language

**XSL** eXtensible stylesheet language

**VSAT** Very Small Aperture Terminal

**GSM** Global System for Mobile Communications

**NCSA** The National Center for Supercomputing Applications

**CERN** European Organization for Nuclear Research

**OSI** Open System Interconnection

**URL** Uniform Resource Locator

**TCP/IP** Transfer Control Protocol / Internet Protocol

**IP** Internet Protocol

**GHz** gigahertz

**CMUCL** Carnegie-Mellon Common Lisp

**XML** eXtensible Markup Language

**LL** parses input from *left* to right using *leftmost* derivation

**MIME** Multipurpose Internet Mail Extensions

**DSL** Domain-Specific Language

**SLIME** Superior Interaction Mode for Emacs

**SQL** Structured Query Language

**RDBMS** Relational Database Management System

**CGI** Common Gateway Interface

**PHP** PHP: Hypertext Preprocessor

**LSP** Lisp Server Pages

**SSL** Secure Sockets Layer

**asdf** Another System Definition Facility

# Appendix C

# Source code

## C.1 Line counts

| Filename | Lines of code | Lines of comments | Empty lines | Line-count |
|---|---|---|---|---|
| utilities.lisp | 150 | 14 | 40 | 204 |
| bnf.lisp | 517 | 23 | 64 | 604 |
| meta.lisp | 102 | 11 | 20 | 133 |
| http.lisp | 817 | 63 | 118 | 998 |
| date-parser.lisp | 99 | 2 | 12 | 113 |
| md5.lisp | 152 | 9 | 11 | 172 |
| server-process.lisp | 54 | 3 | 11 | 68 |

## C.2 Call graphs

The figure C.1 illustrates the calls for functions in the utilities package.

Figure C.3 shows the bnf package. In this one, the main macro is with-bnf, which expands code in the http package.

Figure C.4 shows the calls in the http package. The figure is a little too complex to make sense in print, but should be readable in the electronic version.

I looped over the symbols of each package, looking for the functions. For each function, I asked for the functions that called it. Inverting this relation, I got a graph of the calls made by each function. These are illustrated in two figures, figure C.3 which describes the packages `utilities`, `meta` and `bnf`, and figure C.4 which describes the packages `http` and `server`.

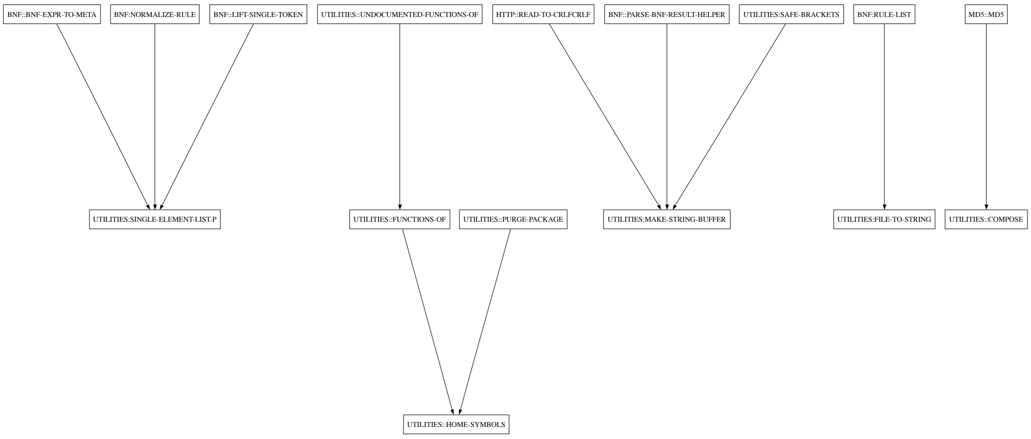

Figure C.1: Functions in the utilities package.

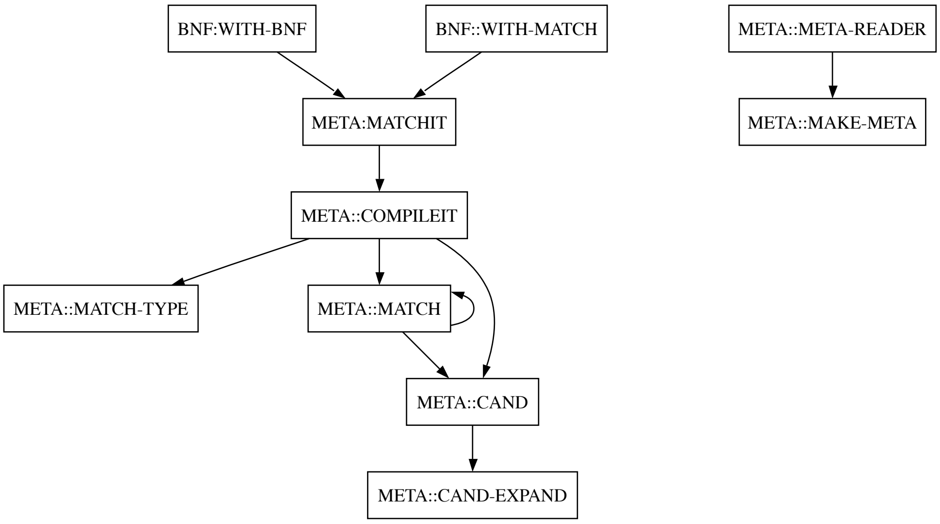

Figure C.2: Functions in the meta package.

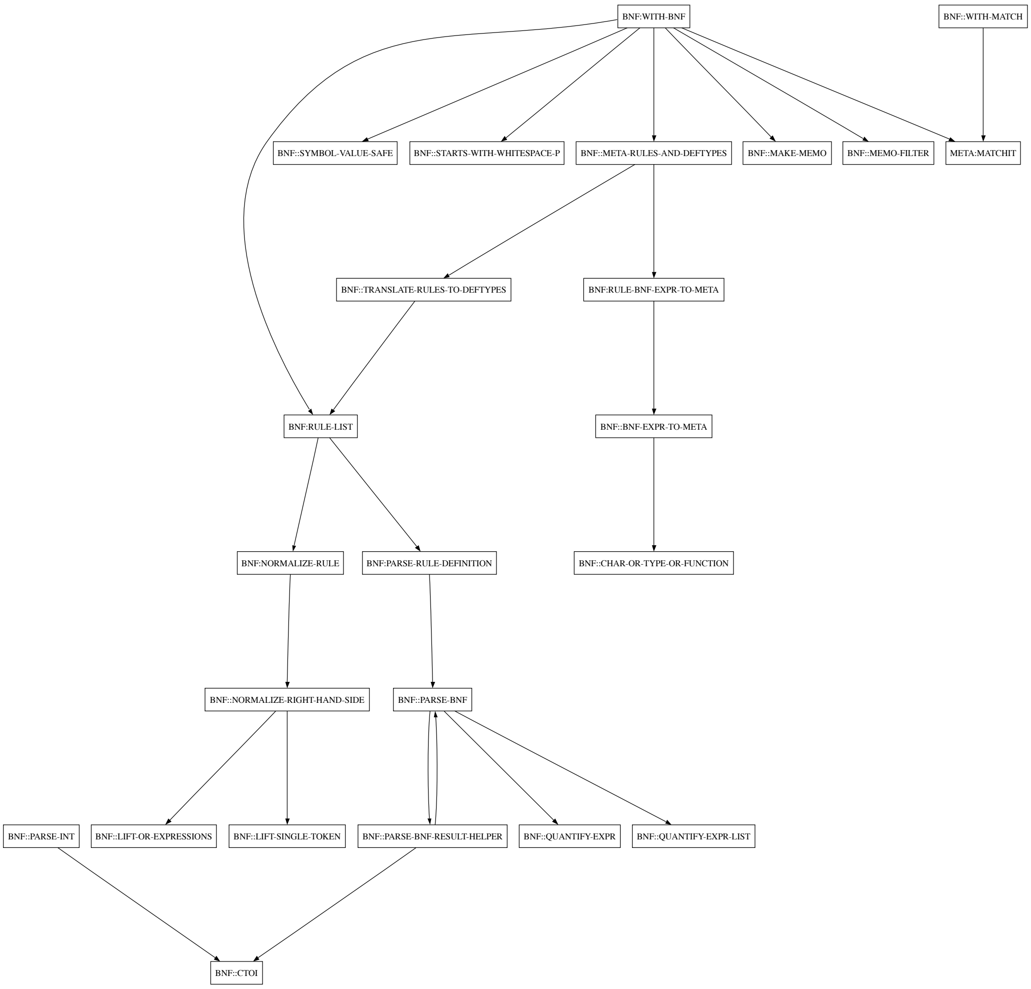

Figure C.3: Functions in the bnf package.

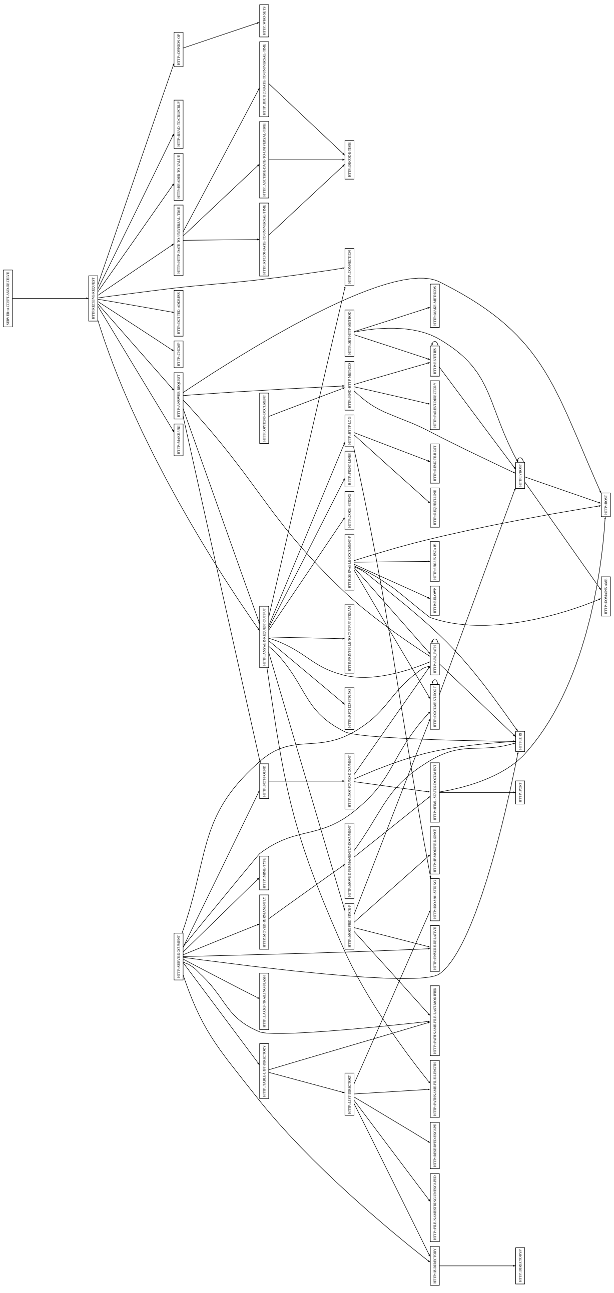

Figure C.4: Functions in the http package.

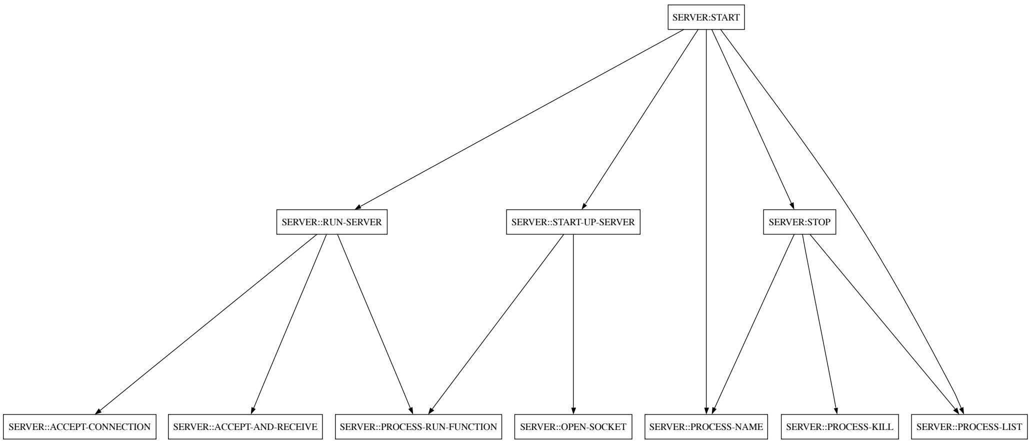

Figure C.5: Functions in the server package.

## C.3 Utilities

```
(defpackage :utilities (:nicknames :util)
  (:use :common-lisp))

(in-package :utilities)


;; used in bnf.lisp
(defmacro maybe (thunk variable)
  "If thunk returns a non-nil value, assign that value to variable."
  (let ((g (gensym)))
    `(let ((,g (funcall
                (function ,thunk))))    ;because we want
                                        ;the lexical value of func
      (when ,g (setf ,variable ,g)))))

;; not used
(defun unshift (list element)
  "If element, add it as the last element of list"
  (when element
    (rplacd (last list) (list element)))
  list)




;; not used
(defmacro apush (key value place)
  "Cons key and value together and cons it onto place."
  (multiple-value-bind (vars forms var set access)
      (get-setf-expansion place)
    `(let* (,@(mapcar #'list vars forms)
            (,(car var) (acons ,key ,value ,access)))
      ,set)))


;; not used
(defun dotted-list-p (list)
  "dotted list n. a list which has a terminating atom that is not nil.
(An atom by itself is not a dotted list, however.)"
  (and (consp list)
       (let ((cdr (cdr list)))
         (or (and (atom cdr)
                  (not (null cdr)))
             (dotted-list-p cdr)))))

;; not in use
(defun safe-brackets (string)
  "Encode html-tags in string."
  (loop for char across string
        with result = (make-string-buffer)
        for replacement = (cdr (assoc char
                                      '((#\< . "&lt;")
                                        (#\> . "&gt;"))))
        if replacement
        do (loop for char across replacement
                 do (vector-push-extend  char result))
        else
        do (vector-push-extend  char result)
        finally (return result)))


;; not in use
(defun print-characters (character-list &key (stream nil) (columns 8))
  "Print a list of characters in pretty columns."
  (format stream (concatenate 'string "~{~"
                              (format nil "~D" columns)
                              "@{\"~C\"~^ | ~}~%~}") character-list))
```

### Debug support, general macros

```
(defvar *dbg* nil "Whether to print debug messages.")

;; used in http.lisp

(defun dbg (string &rest args)
  "Print a debug message string."
  (when *dbg* (apply 'format *debug-io* string args))
  (force-output *debug-io*)
  t)

(defparameter *whitespace-bag*
  (list #\Space #\Tab #\Newline #\Return)
"List of whitespace characters.")

;; used once in http.lisp, once in bnf.lisp
```

```
(defun make-string-buffer (&optional (length 0))
  "Return an adjustable vector of characters with a fill-pointer."
  (make-array length
              :adjustable t
              :element-type 'character
              :fill-pointer 0))

;; used once in http.lisp, once in bnf.lisp
(defun file-to-string (file)
  "Return contents of file as string."
  (with-open-file (in file)
    (let ((buf (make-array (file-length in) :element-type 'character)))
      (read-sequence buf in)
      buf)))

;; used in bnf.lisp
(defun single-element-list-p (list)
  "Does list consist of a single element?"
  (and (consp list)
       (null (cdr list))))

;; used once in http
(defun split-string (string)
  "A list of the words in string."
  (loop for mark = 0 then (1+ point)
        for point = (position #\Space string :test #'char= :start mark)
        collect (subseq string mark point)
        while point))

(defun seq (start end)
  "The numbers from start to end."
  (loop for i from start to end
        collecting i))

(defun char-seq (start end)
  "The characters from start to end."
  (let ((nstart (if (characterp start) (char-code start) start))
        (nend   (if (characterp end) (char-code end) end)))
  (loop for i from nstart to nend
        collect (code-char i))))

(defun home-symbols (designator
                     &aux (package (find-package designator))
                     (result ()))
  "The symbols for whom DESIGNATOR is the home package."
    (do-symbols (sym package result)
      (when (eq (symbol-package sym) package)
        (push sym result))))


(defun purge-package (name &aux (package (find-package name)))
  "Unintern all symbols for whom NAME is the home package."
  (mapcar (lambda (sym) (unintern sym package))
          (home-symbols package)))

(defmacro let-if ((name test) then &optional else)
  "If test evaluates to a non-nil value, bind name to it,
and execute the form then, otherwise execute the form else."
  (let ((g (gensym)))
    `(let* ((,g ,test)
            (,name ,g))
      ,(if else
           `(if ,g ,then ,else)
           `(when ,g ,then)))))

(defun functions-of (designator
                     &aux (package (find-package designator)))
  "List functions whose home package is designator."
  (remove-if (complement #'fboundp) (home-symbols package)))

(defun undocumented-functions-of (designator)
  "List undocumented functions in package designator"
  (remove-if (lambda (f)
               (documentation f 'function))
             (functions-of designator)))


(defun delete-in-tree (item tree)
  "Remove nodes that are eq to ITEM."
  (labels ((rec (tree)
             (if (atom tree)
                 tree
                 (if (eq item (car tree))
                     (rec (cdr tree))
                     (cons (rec (car tree))
                           (rec (cdr tree)))))))
    (rec tree)))


(defun delete-in-tree (itemlist tree)
  "Remove nodes that are eq to ITEM."
  (labels ((rec (tree)
             (if (atom tree)
                 tree
                 (if ;(eq item (car tree))
                  (member (car tree) itemlist)
                     (rec (cdr tree))
                     (cons (rec (car tree))
                           (rec (cdr tree)))))))
    (rec tree)))


(defun compose (&rest fns)
  (destructuring-bind (fn1 . rest) (reverse fns)
    #'(lambda (&rest args)
        (reduce #'(lambda (v f)
                    (funcall f v))
                rest
                :initial-value (apply fn1 args)))))




(export '(unshift maybe apush dbg *dbg* make-string-buffer
          file-to-string dotted-list-p
          single-element-list-p safe-brackets
          *whitespace-bag* split-string char-seq let-if))

(provide "utilities")
```

## C.4 Meta

```
(defpackage :meta (:use :common-lisp))
(in-package :meta)


(defvar index 0)

;; uses index and end
(defmacro match (x)
  "Return a form that matches the character or string X."
  (etypecase x
    (character
     ‘(when (and (< index end)
             ;;(util:dbg
             ;; "Trying to match ~:@C to ~:@C~%"
             ;; ’,x (char string index))
             (eql (char string index) ’,x))
       (incf index) (char string (1- index))))
    (string
     ‘(let ((old-index index)) ; ’old-index’ is a lexical variable.
       (or (cand ,@(map ’list #’(lambda (c) ‘(match ,c)) x))
        (not (setq index old-index)))))))

;; uses index and end
(defmacro match-type (x &optional v)
  "Return a form that matches the type X,
and if V is supplied, stores the value of the X in V."
  (if v
      ‘(when (and (< index end) (typep (char string index) ’,x) )
        (setq ,v (char string index)) (incf index) ,v)
      ‘(when (and (< index end) (typep (char string index) ’,x) )
        (incf index) (char string (1- index)))))

(defstruct (meta
             (:print-function
              (lambda (m s d
                         &aux (char (meta-char m))
                         (form (meta-form m)))
                (declare (ignorable d))
                (ecase char
                  ((#\@ #\! #\$) (format s "~A~S" char form))
                  (#\[ (format s "[~{~A~^ ~}]" form))
                  (#\{ (format s "{~{~A~^ ~}}" form))))))
  "A meta character and the form it precedes."
  char
  form)

(setf (documentation 'meta::meta-char 'function)
      "The read macro character of a meta expression."
      (documentation 'meta::meta-form 'function)
      "The form of a meta expression.")

(defun cand-expand (args result)
  "Helper for the concatenating AND operator, cand."
  (if (null args)
      (cons 'list (nreverse result))
      (let ((gtest (gensym)))
        `(let ((,gtest ,(car args)))
          (when ,gtest
            ,(cand-expand (cdr args) (cons gtest result)))))))

(defmacro cand (&rest args)
  "Concatenating AND.  If all args evaluate to a non-nil value,
collect their values and return that list."
  (if args
      (cand-expand args ())
      :empty))


(defun compileit (x)
  "Given a meta struct of character and form,
return lisp code that correspond to the semantics
of character in the meta language."
  (typecase x
    (meta
     (ecase (meta-char x)
       (#\! (meta-form x))
       (#\[ `(let ((old-index index))
              (or (cand ,@(mapcar #'compileit
                                  (meta-form x)))
               (not (setf index old-index)))))
       (#\{ `(or ,@(mapcar #'compileit (meta-form x))))
;       (#\$ `(not (do ()((not ,(compileit (meta-form x)))))))
       ;; Kleene-star will always return true, I guess.
       (#\$ (let ((value (gensym))
                  (result (gensym)))
              `(do ((,value ,(compileit (meta-form x))
                     ,(compileit (meta-form x)))
                    ,result)
                ((not ,value) (or (nreverse ,result)
                               :kleene-star-empty))
                (push ,value ,result))))
       (#\@ (let ((f (meta-form x)))
              (if (consp f) `(match-type ,(car f) ,(cadr f))
                  `(match-type ,f))))))
    (t `(match ,x))))


(defmacro matchit (x)
  "Call compileit with an unevaluated arg."
  (compileit x))

(eval-when (:compile-toplevel :load-toplevel :execute)

  (defun meta-reader (s c)
    "Read a character and a form and make a meta struct from it."
    (make-meta :char c :form (read s)))

  (mapc #'(lambda (c) (set-macro-character c #'meta-reader))
        '(#\@ #\$ #\!))

;; reader macro character for a sequence
  (set-macro-character
   #\[
   #'(lambda (s c)
       (make-meta :char c
                  :form (read-delimited-list #\] s t))))

;; reader macro character for alternatives
  (set-macro-character
   #\{
   #'(lambda (s c) (make-meta :char c
                              :form (read-delimited-list #\} s t))))

;; reader macro character for end of sequence or alternative
  (mapc #'(lambda (c)
            (set-macro-character c (get-macro-character #\) nil)))
        '(#\] #\}))
  )
(export '(matchit index end))

(provide "meta")
```

## C.5 BNF

```
(require "meta")

(defpackage :bnf (:use :meta :util :common-lisp)
  (:import-from :meta meta:matchit meta:index meta:end)
  (:documentation "To generate parser code from a collection of BNF
  statements."))

(in-package :bnf)

(defvar *debug-indent* -1
"How much to indent the next debug statement.
Used to produce debug statements indented
according to their relationship.")

;; reading numbers

(deftype digit () "The numbers"
‘(member ,@(util::char-seq #\0 #\9)))

(defun ctoi (d)
"From a character to the corresponding number."
       (- (char-code d) #.(char-code #\0)))

(defun parse-int (string &optional (index 0) (end (length string))
                         &aux (s +1) d (n 0))
  "From a string representing a number to the number itself."
 ;;; Lexical ’string’, ’index’, and ’end’, as required by matchit.
 (and
  (matchit
   [{#\+ [#\- !(setq s -1)] []}
    @(digit d) !(setq n (ctoi d))
    $[@(digit d) !(setq n (+ (* n 10) (ctoi d)))]])
  (* s n)))

;; some types

(deftype upalpha () "The uppercase ASCII letters"
  ‘(member ,@(util::char-seq #\A #\Z)))

(deftype loalpha () "The lowercase ASCII letters"
‘(member ,@(util::char-seq #\a #\z)))
```

```
(deftype alpha () "The ASCII letters"
         '(or loalpha upalpha))

(deftype alphanum () "ASCII letters and the digits"
         '(or alpha digit))

(deftype octet () "Any byte"
         '(unsigned-byte 8))

(deftype ctl ()
  "CTL            = <any US-ASCII control character
                    (octets 0 - 31) and DEL (127)>"
  `(member ,@(char-seq (code-char 0) #\Space) #\Delete))

(deftype separator ()  "Character that can't be part of a token."
  '(member #\(  #\)  #\<  #\>  #\@
           #\,  #\;  #\:  #\\  #\"
           #\/  #\[  #\]  #\?  #\=
           #\{  #\} #\Space #\Tab))

(deftype token-char () "Character that can be part of a token."
         '(not (or separator ctl)))

(deftype not-newline () "Anything but a newline."
         '(not (member #\Newline)))

(deftype literal-constituent () "Character inside quotation marks."
  '(and character (not (member #\" #\\))))

;; end of type declarations for bnf.

(defmacro with-match (form &body body) "If FORM matches, eval BODY."
  `(or (and  (matchit ,form) ,@body)
    (progn (setf index old-index) nil)))

(defun parse-bnf (string
                  &optional (index 0) column-start
                  (end (length string))
                  &key (package *package*))
  "Parse right hand side of a rule definition
in bnf format in string into s-expressions.
A version with macros."
```

```
  (values
   (loop
       with l = 0
       and m = t
       for symbol = nil
       for result = nil
       until (= index end)
       do (multiple-value-setq (index  l m symbol result)
            (parse-bnf-result-helper
             string index column-start end package l m symbol))
       while result
       collect (if symbol
                   (quantify-expr-list l m result)
                   (quantify-expr l m result)))
   index))

(defun parse-bnf-result-helper (string index column-start
                                       end package l m symbol
                                       &aux thing)
  (labels
      ((integer (&aux (old-index index) (s +1) d (n 0))
         (declare (ignorable s))
         "Read one or more digits, and parse them as a base-10 integer."
         (with-match [{#\+ [#\- !(setq s -1)] []}
                     @(digit d) !(setq n (ctoi d))
                     $[@(digit d) !(setq n (+ (* n 10) (ctoi d)))]]
                         n))

       (name (&aux (old-index index))
             "Read one or more token-chars, return a symbol"
             (with-match [@token-char $@token-char
                         {[#\: #\: @token-char $@token-char][]}]
                         (let ((*package* (find-package package))
                               (*readtable* (copy-readtable nil)))
                           (setf (readtable-case *readtable*) :preserve
                                 thing
                                 (read-from-string
                                  (subseq string old-index index))))))
           (literal (&aux (old-index index)
                          c
                          (buf (make-string-buffer)))
             "Read a string enclosed in double quotes (\")."
             (with-match ${[@(literal-constituent c)
                         !(vector-push-extend c buf)]
                         [#\\ @(character c)
                         !(vector-push-extend c buf)]}
                         (setf thing buf)))

       (sequence ()
             "Recursive call to parse-bnf returns a sequence."
             (multiple-value-bind (result new-index)
                 (parse-bnf string index column-start end
                            :package package)
               (and result
                    (setf thing result
                          index new-index))))

       (comment (&aux (old-index index))
             "Read and discard up to a newline."
             (with-match $@not-newline))

       (sufficient-indentation (&aux (old-index index)
                                     beginning-of-line)
             "Read whitespace until we are at column COLUMN-START."
             (with-match
                 [!(setf beginning-of-line index)
                 $#\Space
                 !(>= (- index beginning-of-line) column-start)])))
    (matchit
     ;; read prefix
     [ $#\Space
       $[#\; !(comment)]
       {[#\Newline !(not (sufficient-indentation))]
        [{[#\Newline !(sufficient-indentation)] []}
         $#\Space
         {[!(maybe integer l)
            {[{#\*[#\# !(setf symbol #\#)]}
              {[!(maybe integer m)]
               [!(setf m t)] }]
             !(setf m l) } ]
          [ {#\*[#\# !(setf symbol #\#)]}
            !(setf l 0)
            {!(maybe integer m) !(setf m t)} ]
          [!(setf l 1 m 1)] }

         ;; read thing
```

```
         {[#\" !(literal) #\" ]
          [#\( !(sequence)  #\) ] ;concat
          [#\; !(comment)  ] ;comment
          [#\[ !(sequence)  #\] !(setf l 0 m 1)] ;optional
          [#\| !(setf thing ’or-token) ] ;alternate
          [#\< #\" #\> !(setf thing "\"")]
          !(name)
          []}
        ]
       }
     ]))
  (values index l m symbol thing))



(defun parse-rule-definition (string
                              &optional (index 0)
                              &key (package *package*)
                              (end (length string))
                              &aux (old-index index)
                              read-list rule  name
                              parse-bnf beginning-of-line
                              column-start)
  "Parse right hand side of a rule definition
in bnf format in string into s-expressions."
  (declare (ignorable read-list rule))
  (if (= index end)
      (values nil index)
      (labels ((name (&aux (old-index index))
                 (with-match [@token-char $@token-char
                             {[#\: #\: @token-char $@token-char][]}]
                             (let ((*package* (find-package package))
                                   (*readtable* (copy-readtable nil)))
                               (setf
                                (readtable-case *readtable*) :preserve
                                name (read-from-string
                                      (subseq string
                                              old-index index))))))
               (comment (&aux (old-index index))
                 (with-match $@not-newline)))
        (with-match [$[$#\Space
                       {[#\; !(comment)] []}
                       #\Newline]
```

```
                      [!(setf beginning-of-line index)
                        $#\Space
                        !(name)
                        $#\Space !(setf column-start
                                        (- index beginning-of-line))
                        #\=
                        $#\Space
                        !(multiple-value-setq (parse-bnf index)
                           (parse-bnf string index
                                      column-start end
                                      :package package))]])
        (values (when name (list ’rulename name parse-bnf))
                index))))

(defun normalize-rule (rule)
  "Lift single elements in rule."
   (let ((rhs (normalize-right-hand-side (caddr rule))))
     (when (single-element-list-p rhs)
       (setf rhs (car rhs)))
     ‘(rulename ,(second rule)
       ,rhs)))

(defun lift-single-token (tree)
  "Return a tree in which single element lists
have been replaced by the element itself."
  (if (atom tree)
      tree
      (if (single-element-list-p (car tree))
          (lift-single-token (cons (caar tree) (cdr tree)))
          (cons (lift-single-token (car tree))
                (lift-single-token (cdr tree))))))

(defun normalize-right-hand-side (expr)
  "Translate single element lists, and normalize alternatives."
  (lift-or-expressions (lift-single-token expr)))

(defun lift-or-expressions (expr)
  "Normalize alternatives."
  (if (atom expr)
      expr
      (if (eq ’or-token (second expr))
          (let ((position (position ’or-token expr)))
            (cons ’parsed-or-expression
                  (cons (lift-or-expressions
                         (if (= position 1) (car expr)
                             (subseq expr 0 position)))
                        (remove ’parsed-or-expression
                                (lift-or-expressions
                                 (subseq expr (1+ position)))))))
          (cons (lift-or-expressions (first expr))
                (lift-or-expressions (rest expr))))))

(defun bnf-expr-to-meta (expr)
  "Convert a bnf expression to a semantically equivalent
expression in meta."
  (if (atom expr)
    (cond
      ((equal expr ’the-empty-string) "[]")
      ((stringp expr)
       (if (= (length expr) 1)
           (format nil "~@C" (char expr 0))
         (format nil "~S"  expr)))
      (t (or (format nil "~A" (char-or-type-or-function expr)) "")))
    (let ((rest (cdr expr)))
      (case (car expr)
        (parsed-or-expression
         (format nil "{~{~A~^ ~}}" (mapcar #’bnf-expr-to-meta rest)))
        (kleene-star
         (let ((result (bnf-expr-to-meta rest)))
           (format nil (if (listp result)
                           "$[~{~A~^ ~}]"
                           "$~A") result)))
        (name (char-or-type-or-function (car rest)))
        (type (destructuring-bind (type symbol)
                  rest
                (format nil "@(~A ~A)" type symbol)))
        (comment "")
        (t
         (if (single-element-list-p expr)
             (format nil "~A" (bnf-expr-to-meta  (car expr)))
             (format nil "[~{~A~^ ~}]"
                     (mapcar #’bnf-expr-to-meta expr))))))))

;; list of possible types:
;; the-empty-string
;; a string "string" (a literal)
;; a single-character string "s"
;; a list == sequence
;; a list with a "magic" car such as
;; (parsed-or-expression kleene-star
```

End Generic character sets

```
(declaim (notinline char-or-type-or-function))

(defun char-or-type-or-function (name)
  "Name is a character, type or form to be evaluated.
Return corresponding meta expression."
  (declare (special deftypes))
  (format
   nil
   (cond
     ((characterp name) "~@C" )
     ((or #+openmcl (ccl:type-specifier-p name)
          #+lispworks (SYSTEM::VALID-TYPE-SPECIFIER name)
          #+cmu   (handler-case (typep nil name)
                    (simple-error () nil)
                    (:no-error (ignore) (declare (ignore ignore)) t))
          (find name deftypes :key #'second)
          ) "@~S")
     (t "!(~S)"))
   name))

(defun quantify-expr (min max body)
  "Deal with the sequence quantifiers of BNF."
  (if (zerop min)                        ; expr    min     max
      (if (integerp max)                ; *M      0       int
          (labels ((optionals (count)
                     (cond ((> 1 count) nil)
                           ((= 1 count)
                            `((parsed-or-expression
                               ,body the-empty-string)))
                           (t
                            (list (list 'parsed-or-expression
                                        (cons body
                                              (optionals (1- count)))
                                          'the-empty-string))))))
            (optionals max))
          (list (funcall  (if (listp body)
                              #'cons #'list)
```

```
                          'kleene-star body))) ; N*M     plusp   int /=
      (append
       (loop for i below min
             collect body)
       (quantify-expr 0 (if (integerp max)(- max min) t) body))))


(defun quantify-expr-list (min max body)
  "Deal with the comma-separated list quantifiers of BNF."
  (let* ((comma (list '(kleene-star
                        (kleene-star
                         ((parsed-or-expression #\Return #\Newline)
                          (parsed-or-expression #\Space  #\Tab)))
                        (parsed-or-expression #\Space #\Tab))
                      #\,))
         (commas (list (cons 'kleene-star comma)))
         (subsequent (list comma commas body)))
    (labels ((nconc-times (n expr)
               (loop for i below n
                     nconc expr)))
      (list commas
            (if (zerop min)
                (list 'parsed-or-expression body 'the-empty-string)
                (cons body
                      (nconc-times (decf min) subsequent)))
            (if (integerp max)
                (nconc-times (- max min 1)
                             (list
                              (list
                               'parsed-or-expression
                               subsequent
                               commas)))
                (list 'kleene-star subsequent))
            commas))))

(defun rule-bnf-expr-to-meta (rule)
  "Translate the right hand side of rule to META."
  (declare (special deftypes))
  (list 'rulename (cadr rule)
          (bnf-expr-to-meta (caddr rule))))

(defun rule-list (file package)
  "A list of rules as parsed from file."
  (do ((string (file-to-string file))
       (parse t)
       (index 0)
       (ruleset ()))
      ((not parse) ruleset)
    (multiple-value-setq (parse index)
      (parse-rule-definition string index :package package))
    (when parse
      (push (normalize-rule parse) ruleset))))
```

```
(defun meta-rules-and-deftypes (rules)
  "Split the ruleset into those that are deftypes
and those that are rules."
  (let ((deftypes (bnf::translate-rules-to-deftypes rules)))
    (declare (special deftypes))
    (values (mapcar #'rule-bnf-expr-to-meta
                    (set-difference rules deftypes :key #'second))
            deftypes)))

(defun translate-rules-to-deftypes (&optional
                                    (rules (bnf::rule-list
                                            "sample-rules.txt"
                                            :rfc2616))
                                    &aux deftypes)
  "Return deftype statements for the rules in rules
that can be translated."
  (labels ((translate-to-deftype
               (rule &aux (translation
                           (bnf::translate-to-deftype-args
                            (third rule))))
             (when translation
               `(deftype ,(second rule) ()
                  ',(typespec-args translation))))
           (typespec-args (args)
             (when (atom args) (setf args (list args)))
             (let* ((symbols (remove-if (complement #'symbolp) args))
                    (characters (remove-if (complement #'characterp)
                                           args))
                    (member-statement (when characters
                                        (cons 'member characters))))
               (if symbols
                   (if member-statement
                       (list* 'or member-statement symbols)
                       (cons 'or symbols))
                   member-statement)))
           (translate-to-deftype-args (rhs)
             (typecase rhs
               (string (when (and (array-in-bounds-p rhs 0)
                                  (not (array-in-bounds-p rhs 1)))
                         (char rhs 0)))
               (list (and (eq (car rhs) 'bnf::parsed-or-expression)
                          (let ((rest
                                 (mapcar #'translate-to-deftype-args
                                         (cdr rhs))))
                            (when (every (complement #'null) rest)
                              rest))))
               (symbol
                (when (or (find rhs deftypes :key #'second)
                          (let ((rule (find rhs rules :key #'second)))
                            (when rule
                              (translate-to-deftype rule))))
                  rhs)))))
    (dolist (rule rules)
      ;; the rule may have occured in an earlier definition
      ;; and thus be defined already
      (unless (find (second rule) deftypes :key #'second)
      (let ((translation (translate-to-deftype rule)))
        (when translation
          (format *standard-output* "~A~%" translation)
          (push translation deftypes)))))
    deftypes))

(declaim (notinline starts-with-whitespace-p))

(defun starts-with-whitespace-p (name whitespace-list)
  "True if name is listed amongst the names that need
their preceding whitespace in order to match as intended.
Whitespace-list is either a list of symbols or a symbol
whose value is such a list."
  (member name (if (listp whitespace-list) whitespace-list
                   (symbol-value whitespace-list))))


(defstruct rulematch "Rule is the name of the rule,
string is what it matched,
and matches is a list of the rules
that were used to match the rule."
rule string matches)


(defparameter *memos* ())

(defstruct memo
    pathname
    mtime
    whitespace-list
    package
    labels-part
    deftypes
    names)

(defun memo-filter (pathname mtime whitespace-list package)
  "Return a function that returns the argument, a memo, if the slots
of the memo match the four arguments."
  (lambda (memo)
    (when (and (equal pathname (memo-pathname memo))
               (= mtime (memo-mtime memo))
               (equal whitespace-list (memo-whitespace-list memo))
               (equal package (memo-package memo)))
      memo)))

(defun symbol-value-safe (sym)
  (if (symbolp sym)
      (symbol-value sym)
      sym))

(defmacro with-bnf (file whitespace-list package &body body)
  "Generate a LABEL form based on the bnf in FILE,
and eval BODY within it.
Whitespace-list is the names that need their
preceding whitespace in order to match as intended.
Package is a preferably empty package where
we'll intern the symbols read from FILE.
Return a hierarchy describing the matches."
    (let* ((file (probe-file file))
           (memo (find-if
                  (memo-filter file
                               (file-write-date file)
                               (symbol-value-safe whitespace-list)
                               (find-package package)) *memos*)))
      (if memo
          `(let ,(memo-names memo)
            ,@(memo-deftypes memo)
            (labels ,(memo-labels-part memo)
              ,@body))
      (let  ((names ())
             (labels-part ()))
      (multiple-value-bind (meta-rule-list deftypes)
          (meta-rules-and-deftypes (rule-list file package))
        (loop for (nil name definition) in meta-rule-list
              do (push
                  `(,name (&aux (old-index index) c)
                    (declare (ignorable c) (special index c))
                    (or (and
                         (dbg "~v,8TTrying to match ~A.~@?"

                              (incf *debug-indent*)
                              ',name
                              " (String is at: ~S)~%"
                              (subseq string index
                                      (min end (+ 20 index))))
                         ,(if (bnf::starts-with-whitespace-p
                               name whitespace-list) t
                               `(matchit ,(read-from-string
                                           "${#\\Space #\\Tab}")))
                         (let ((past-whitespace index))
                           (let ((match
                                  (matchit
                                   ,(read-from-string
                                     (if (bnf::starts-with-whitespace-p
                                          name whitespace-list)
                                         definition
                                         (format nil
                                                 "~A" definition))))))
                             (and match
                                  (dbg "~v,8TFound the match ~A.  ~@?"
                                       (shiftf *debug-indent*
                                               (1- *debug-indent*))
                                       ',name
                                       "It's: ~S.  Match is ~A~%"
```

```
                                       (subseq string old-index index)
                                       match)
                                  (make-rulematch
                                   :rule ',name
                                   :string (car (push
                                                 (subseq string
                                                         past-whitespace
                                                         index) ,name))
                                   :matches (util::delete-in-tree
                                             (list :kleene-star-empty
                                                   :empty) match))))))
                     (progn (setf index old-index)
                            (dbg "~v,8TFailed to match ~A.~%"
                                 (shiftf *debug-indent*
                                         (1- *debug-indent*)) ',name)
                            nil)))
                  labels-part)
              do (push name  names))
        (push (make-memo :pathname file
                         :mtime (file-write-date file)
                         :whitespace-list
                         (symbol-value-safe whitespace-list)
                         :package (find-package package)
                         :deftypes deftypes
                         :labels-part labels-part
                         :names names)
              *memos*)
        `(let ,names
          ,@deftypes
          (labels ,labels-part
            ,@body))))))

(export '(parse-rule-definition normalize-rule rule-bnf-expr-to-meta
          with-bnf rule-list ))

(provide "bnf")

;; local variables:
;; eval: (modify-syntax-entry ?{  "(}  ")
;; eval: (modify-syntax-entry ?}  "){  ")
;; eval: (modify-syntax-entry ?\[ "(]  ")
;; eval: (modify-syntax-entry ?\] ")[  ")
;; end:
```

## C.6 http

```
(defpackage :http
    (:use :common-lisp :util :bnf :meta))

#+lispworks (require "lispworks-http" "compat/lispworks/http.lisp")
#+openmcl (require "openmcl-http" "compat/openmcl/http.lisp")
#+cmu (require "cmucl-http" "compat/cmucl/http.lisp")

(in-package :http)

(require "date-parser" '("date-parser"))

(eval-when (:compile-toplevel)
  (load "date-parser"))

(require "bnf" '("bnf"))
```

### C.6.1 Configuration

```
;; other strings that may go here: machine-type, machine version,
;; machine-instance, lisp-implementation-type,
;; lisp-implementation-version , short-site-name, long-site-name

(defvar *server-identification* (format nil "Clack/0.1 (~A)"
                                        (software-type))
  "Product tokens are used to allow communicating applications
to identify themselves by software name and version.")
```

### C.6.2 Formatted strings

```
;; would be better in date-parser.lisp??
(defun rfc1123-string (&optional (time (get-universal-time)))
  "Format date like Sun, 06 Nov 1994 08:49:37 GMT"
  (multiple-value-bind (second minute hour date
                               month year day)
      (decode-universal-time time 0)
    (format nil "~@?, ~2,'0D ~@? ~4,'0D ~2,'0D:~2,'0D:~2,'0D GMT"
            #.(indexed-format
               (mapcar (lambda (name) (subseq name 0 3 ))
                       *weekdays*))
            day date
            #.(indexed-format *months*) month
```

```
           year hour minute second)))
```

### C.6.3 Utility functions

```
(defun print-lines (os &rest args)
  (dolist (line args)
    (princ line os)
    (princ #\Return os)
    (terpri os)))
```

### C.6.4 URI

```
(defclass uri ()
  ((uri-reference :type string
                  :accessor uri-reference :initarg :uri-reference)
   (abs_path :accessor abs_path :initarg :abs_path)
   (scheme :accessor scheme :type string :initarg :scheme)
   (authority :accessor authority :type string :initarg :authority)
   (userinfo :accessor userinfo :type string :initarg :userinfo)
   (host :accessor host :type string :initarg :host)
   (port :accessor port :initform "80" :type string :initarg :port)
   (path :accessor path :type string :initarg :path)
   (query :accessor query :type string :initarg :query)
   (fragment :accessor fragment :type string :initarg :fragment))
  (:documentation "A uniform resource identifier."))

(defun make-uri (uri-reference abs_path scheme authority
                              userinfo host port path
                              query fragment)
  "Return a uri object."
  (make-instance 'uri
                 :uri-reference (or uri-reference "")
                 :abs_path (or abs_path "")
                 :scheme (or scheme "")
                 :authority (or authority "")
                 :userinfo (or userinfo "")
                 :host (or host "")
                 :port (or port "80")
                 :path (or path "")
                 :query (or query "")
                 :fragment (or fragment "")))
```

### C.6.5 Packages for rulesets

```
;; we do not want ruleset to inherit from any default package
;; From the Hyperspec:
;; "If :use is not supplied, it defaults to the same
;;  implementation-dependent value as the :use argument to make-package."

(defpackage #:rfc2616 (:use)
  (:documentation "Holds rule definitions from RFC 2616.
  (\"Hypertext Transfer Protocol -- HTTP/1.1\")"))
(defpackage #:rfc1123 (:use)
  (:documentation "Holds rule definitions from RFC 1123.
  (\"Requirements for Internet Hosts -- Application and Support\")"))
(defpackage #:rfc822  (:use)
  (:documentation "Holds rule definitions from RFC 822.
  (\"Standard for the Format of ARPA Internet Text Messages\")"))
(defpackage #:rfc2109 (:use)
  (:documentation "Holds rule definitions from RFC 2109.
  (\"HTTP State Management Mechanism\")"))
(defpackage #:rfc2396 (:use)
  (:documentation "Holds rule definitions from RFC 2396.
  (\"Uniform Resource Identifiers (URI): Generic Syntax\")"))
```

### C.6.6 Presets for ruleset

```
(deftype rfc2616::octet () "Any byte" '(character) )

; interesting choice of docstring,
; as if readers prefer binary ops,
; while compilers prefer less-than
(defun rfc2616::ascii-char-p (char)
  "False if the high bit of character CHAR is set."
       (< (char-code char) 128))

(deftype rfc2616::char ()
  "High bit not set"
  '(and character (satisfies ascii-char-p)))

(deftype rfc2616::|not-newline| ()
         "Anything but a newline"
         '(not (member #\Newline #\Return)))
(deftype rfc2616::|horizontal-whitespace| ()
         "Space or tab"
         '(member #\Space #\Tab))
```

```
(defstruct rulematch
  "Rule is the name of the rule, string is what it matched,
and matches is the \"breadcrumb\" path that led
to the rule being called."
  rule string matches)

(defun who-sets (item tree &aux (result ()))
  "the paths that lead to item in tree."
  (labels ((rec (tree path)
             (typecase tree
               (list
                (loop for subtree in tree
                      do (rec subtree path)))
               (bnf::rulematch
                (if (eq (rulematch-rule tree) item)
                    ;; will not look for further matches down this path
                    (push (cons (rulematch-string tree) path) result)
                    (rec (rulematch-matches tree)
                         (cons (rulematch-rule tree) path)))))))
    (rec tree ()))
  result)

(defun opinion-of (whom regarding  where)
  "Return the first opionion of WHOM
as to the value of REGARDING
based on the set of matches in WHERE."
  (labels ((relevant (reason)
             (member whom reason)))
    (find-if #'relevant (who-sets regarding where))))

(defparameter *last-match* nil)


;; this has to be a list of symbols, as these do not yet have values.
(eval-when (:compile-toplevel :load-toplevel :execute)
  (defparameter *whitespace-list*
    '(RFC2616::|star-lws| RFC2616::SP RFC2616::HT RFC2616::LF
      RFC2616::CR RFC2396::|segment| RFC2396::|toplabel| RFC2396::|uric|
      RFC2396::|domainlabel| RFC2396::|escaped| RFC2396::|reserved|
      RFC2396::|pchar| RFC2396::|unreserved| RFC2396::|alphanum|
      RFC2396::|path_segments| RFC2616::|lowalpha| RFC2616::|upalpha|
      RFC2616::|alpha| RFC2616::|digit| RFC2616::|token-char|)))
```

```
(defun read-to-crlfcrlf (is)
  "Collect characters from input stream IS
until we find a sequence of CRLFCRLF."
  (loop with state = 0
         with buflength = 1024
         with string = (util::make-string-buffer buflength)
         with state-char = #(#\Return #\Newline #\Return #\Newline)
         for char = (read-char is nil nil)
         while char
         do (vector-push-extend char string buflength)
         if (char= char (svref state-char state))
         do (when (= (incf state) 4) (return string))
         else
         do (setf state 0)))

(defmacro generate-headers (&rest name-and-values)
  "Return a list of header lines."
  (cons ’list
         (loop for nv in name-and-values
               when nv
               collect (if (and (stringp nv) (string= nv ""))
                           nv
                           (if (car nv) ‘(concatenate ’string
                                          ,@(if (stringp (car nv))
                                                ‘(,(concatenate ’string
                                                               (car nv)
                                                               ": "))
                                                ‘(,(car nv) ": "))
                                          ,(cadr nv)))))))

(defun answer-request-output (is request time code mime-type
                                  body last-modified entity bytes-sent)
  "An entity is found, and code and body are prepared.
 We default to handling a HEAD request.
 If it is a GET, add body to the answer."
  (declare (notinline http-log))
  (dbg "a-r-o-args: ~@{~A~^, ~}~%" is request time code mime-type
       body last-modified entity bytes-sent)
  ;; attach another output stream to is
  (apply #’print-lines is
           (format nil "HTTP/1.1 ~D ~A" code (code-string code))
          (macrolet ((when-gh (test (name value))
                      `(when ,test (generate-headers (,name ,value)))))
            (append (generate-headers ("Date" (rfc1123-string))
                                      ("Server" *server-identification*))
                    (when-gh request
                             ("Connection" (if
                                           (string-equal
                                            (connection request)
                                            "close")
                                              "close"
                                              "Keep-Alive")))
                    (when-gh (string-equal (method request) "OPTIONS")
                             ("Allow"
                              (format nil "~{~A~^, ~}"
                                     '(GET HEAD OPTIONS TRACE))))
                    (when-gh (= code 301)
                             ("Location"
                              (concatenate
                              'string
                              (abs_path (uri request)) "/")))
                    (when-gh last-modified
                             ("Last-Modified"
                             (rfc1123-string last-modified)))
                    (when-gh body
                             ("Content-Length"
                              (prin1-to-string
                               (setf
                               bytes-sent
                               (typecase body
                                 (string (length body))
                                 (pathname
                                  (pathname-file-length body)))))))
                    (when-gh mime-type
                             ("Content-Type" mime-type))
                    (list ""))))
  (when (and (string-equal (method request) "GET")
              (modified-since-p request entity))
    (typecase body
      (string   (princ body is))
      (pathname (print-file-to-output-stream body is))))
  (force-output is)
  (http-log request entity time code bytes-sent)
  (values request entity is))
```

```
(defun receive-request (is)
  "Read and handle a request.
 Return false if stream should be closed,
true if the stream should be kept alive."
  (let* ((time (get-internal-run-time))
          (string (read-to-crlfcrlf is)) ; if this times out,
                                        ; we  may send a
                                        ; 408 Request Timeout
          (index 0)
          (end (length string)))
    (when string
      (bnf::with-bnf "sample-rules.txt"
          *whitespace-list*;; a list of symbols
          :rfc2616
        ;;the 'matchit should be wrapped in a timeout for a 408 check
        (let ((tree (matchit !(rfc2616::|HTTP-message|))))
          (setf *last-match* tree)
          (if (null tree)
              ;; requires a different a-r-o
              (answer-request-output is nil time 400
                                     nil nil nil nil nil)
              (string-not-equal
               (connection
                (answer-request
                 is (make-instance
                     'request
                     :request-line (chomp (car rfc2616::|Request-Line|))
                     :remote-host (dotted-address is)
                     :uri (apply
                           #'make-uri rfc2396::|URI-reference|
                           (mapcar #'car
                                   (list (last rfc2396::|abs_path|)
                                         rfc2396::|scheme|
                                         rfc2396::|authority|
                                         rfc2396::|userinfo|
                                         rfc2396::|host|
                                         rfc2396::|port|
                                         rfc2396::|path|
                                         rfc2396::|query|
                                         rfc2396::|segment|)))
                     :host (or
                            (and (opinion-of 'rfc2616::|Request-URI|
                                             'RFC2396::|absoluteURI|
                                             tree)
                                 ;; Request-URI is an absoluteURI.
                                 (car
                                  (opinion-of ’rfc2616::|Request-URI|
                                              ’RFC2396::|host| tree)))
                            ;; request-uri is not an absoluteuri
                            (and rfc2616::|Host|
                                 (car (opinion-of
                                       ’rfc2616::|Host|
                                       ’RFC2396::|host| tree))))
                     ;; should return a 400 Bad Request.
                     ;; unless the HTTP Version is 1.0,
                     ;; then we can guess.
                     :method  (car rfc2616::|Method|)
                     :headers  (append rfc2616::|general-header|
                                       rfc2616::|request-header|
                                       rfc2616::|entity-header|)
                     :http-version (car rfc2616::|HTTP-Version|)
                     :connection  (car rfc2616::|connection-token|)
                     :if-modified-since
                     (and (not
                           (null rfc2616::|If-Modified-Since|))
                          (HTTP-date-to-universal-time
                           (header-to-value
                            (car rfc2616::|If-Modified-Since|))))
                     :body (car rfc2616::|message-body|)) time))
               "Close")))))))
```

```
(defun header-to-value (string)
  "Get the value part of a simple mime-type header."
  (string-trim util:*whitespace-bag*
               (subseq string (1+ (position #\: string)))))

(defparameter *status-codes*
  ’((100 . "Continue")
    (101 . "Switching Protocols")
    (200 . "OK")
    (201 . "Created")
    (202 . "Accepted")
    (203 . "Non-Authoritative Information")
    (204 . "No Content")
    (205 . "Reset Content")
    (206 . "Partial Content")
    (300 . "Multiple Choices")
    (301 . "Moved Permanently")
    (302 . "Found")
    (303 . "See Other")
    (304 . "Not Modified")
    (305 . "Use Proxy")
    (307 . "Temporary Redirect")
    (400 . "Bad Request")
    (401 . "Unauthorized")
    (402 . "Payment Required")
    (403 . "Forbidden")
    (404 . "Not Found")
    (405 . "Method Not Allowed")
    (406 . "Not Acceptable")
    (407 . "Proxy Authentication Required")
    (408 . "Request Time-out")
    (409 . "Conflict")
    (410 . "Gone")
    (411 . "Length Required")
    (412 . "Precondition Failed")
    (413 . "Request Entity Too Large")
    (414 . "Request-URI Too Large")
    (415 . "Unsupported Media Type")
    (416 . "Requested range not satisfiable")
    (417 . "Expectation Failed")
    (500 . "Internal Server Error")
    (501 . "Not Implemented")
    (502 . "Bad Gateway")
    (503 . "Service Unavailable")
    (504 . "Gateway Time-out")
    (505 . "HTTP Version not supported"))
  "Status Code mapped to Reason Phrase."
  )

(defun code-string (code)
  "The Reason Phrase corresponding to the status-code CODE."
  (cdr (assoc code *status-codes*)))

(defun code-and-reason-string (code)
  "The Reason Phrase corresponding to the status-code CODE."
  (destructuring-bind (code . reason) (assoc code *status-codes*)
    (format nil "~A ~A" code reason)))

(defun http-version-not-supported (is request time)
  "The http-version is not supported."
  (declare (ignore is request time))
  (values 505))

(defun not-recognized (is request time)
  "The method is not recognized."
  (declare (ignore is request time))
  (values 501))

(defun not-modified (is request time)
  "The document is not modified, so return the code for that."
  (declare (ignore is request time))
  "The entity is not modified, so we'll inform the client."
  (values 304))

(defun modified-since-p (request entity)
  "Is the entity ENTITY modified since the timestamp in REQUEST?"
  (not  (util::let-if (if-modified-since (if-modified-since request))
            (util::let-if
                (entity-modified (pathname-file-last-modified
                                  (merge-pathnames
                                   (ensure-relative entity)
                                   (document-root request))))
                (=  if-modified-since
                    entity-modified)))))

(defun pathname-file-length (pathname)
  "The length in bytes of the file pointed to by pathname."
  (with-open-file (in pathname) (file-length in)))

(defun pathname-file-last-modified (pathname)
  "When was the file pointed to by PATHNAME last modified."
  (file-write-date pathname))



(defun http-log (request entity time code bytes-sent)
  "Make a log entry."
  (declare (ignorable entity))
  (#_syslog #$LOG_INFO (ccl::make-cstring "%s") ;; Openmcl specific call
            ;; Franz would use
            ;; (excl.osi:syslog *log-info*, hopefully with lisp strings
            ;; for cmucl, look at c-call from the alien package
            :address (ccl::make-cstring
                      (format nil "~A - - [~A] ~S ~D ~D ~D B/s"
                              (remote-host request)
                              (iso-8601-string)
                              (request-line request)
                              code
                              bytes-sent
                              (let ((divisor
                                     (- (get-internal-run-time) time)))
                                (if (zerop divisor)
                                    0
                                    (floor
                                     (* bytes-sent
                                        internal-time-units-per-second)
                                        divisor)))))))

(defun chomp (string)
  "Cut whitespace from end of string."
  (subseq string 0
          (1+ (position-if
               (lambda (char)
                 (not (member char util:*whitespace-bag*)))
               string :from-end t))))

(defun lacks-trailing-slash (namestring)
  "False if namestring ends in slash."
  (loop for char across namestring
        finally (return (char/= #\/ char))))

(defvar *mime-default* "text/plain"
  "If we do not know better, serve file as text.")

(defpackage #:mime (:use)
  (:import-from "BNF" "NOT-NEWLINE" "TOKEN-CHAR")
  (:documentation
   "Package for the rules for how to read the file
 that specifies the suffix-to-mime-type mapping."))

(defparameter *mime-suffix-alist*
  (let* ((string (file-to-string "/private/etc/httpd/mime.types"))
          (index 0)
          (end (length string))
          (collection ()))
    (bnf::with-bnf "mime-filter.bnf"
        (#\Space) :mime
      (loop
           (unless (matchit !(mime::|mime-list|)) (return collection))
           (when (shiftf mime::|type-suffixes| nil)
             (apush (shiftf mime::|suffix| nil) mime::|type| collection))
         (setf mime::|type| nil))))
  "An association list where the first element
is a list of suffix strings  such as (\"jpeg\" \"jpg\"),
and the second element is a mime designator such as \"image/jpeg\".")

(defun mime-type (suffix)
  "Return a mime-type, such as \"image/jpeg\"
that corresponds to a suffix such as \"jpg\"."
  (util::let-if
      (list  (find-if (lambda (suffix-list)
                        (member suffix suffix-list :test #'string=))
                      *mime-suffix-alist* :key #'car))
      (cadr list) *mime-default*))


(defun recognized-method-p (method)
  (member method '("OPTIONS" "GET" "HEAD" "POST"
                   "PUT" "DELETE" "TRACE" "CONNECT")
           :test #'string=))

(defun answer-request (is request time)
  "The request object is prepared, and we should answer it.

 29 lines is a little too much.  What does it do?  It takes
 a request, and prepares headers and body.
 If not found, not modified, or a directory, it will dispatch,
 otherwise it gets values from serve-document.  Then it calls a-r-o.
 "
  ;; if HTTP-Version > 1.1, return "505 HTTP Version Not Supported" ?
  (let ((entity (servable-document-p request))
         (bytes-sent 0))
    (multiple-value-bind
           (code mime-type body last-modified)
         (cond   ((string/= (http-version request) "HTTP/1.1")
                  (http-version-not-supported request is time))
                 ((not (recognized-method-p (method request)))
                  (not-recognized request is time))
                 ((not entity) (not-found request is))
                 ((not (modified-since-p request entity))
                 (not-modified is request time))
                 ((string-equal (method request) "OPTIONS")
                  (options-document request entity is))
                 ((is-directory entity)
                  (if (lacks-trailing-slash (abs_path (uri request)))
                      (moved-permanently
                      request is
                      (concatenate 'string
                                   (abs_path
                                    (uri request)) "/"))
                      (table-list-directory
                      request
                      (make-pathname :defaults entity
                                     :name :wild) is)))
                 (t (serve-document request entity is)))
      (answer-request-output is request time code
                             mime-type body last-modified
                             entity bytes-sent))))

(defclass request ()
  ((headers :accessor headers :initarg :headers)
   (method :accessor method :initarg :method)
   (uri :accessor uri :initarg :uri)
   (host :accessor host :initarg :host)
   (http-version :accessor http-version :initarg :http-version)
   (if-modified-since :accessor if-modified-since
                      :initarg :if-modified-since)
   (body :accessor body :initarg :body)
   (connection :accessor connection :initarg :connection)
   (remote-host :accessor remote-host :initarg :remote-host)
   (request-line :accessor request-line :initarg :request-line))
  (:documentation "Internal representation of a request."))

(defmethod abs_path ((request request))
  "Conveniently ask for the abs_path of the uri of the request."
  (abs_path (uri request)))
```

```
(defun answer-request (is request time)
  "The request object is prepared, and we should answer it."
  ;; if HTTP-Version > 1.1, return
  ;; "505 HTTP Version Not Supported" ?
  ;; -- handle that further up the chain
  ;;
  (let* ((abs_path (abs_path request))
          (handler (find-http-method (method request)
                                     abs_path (host request)))
          (bytes-sent 0))
    (multiple-value-bind
          (code mime-type body last-modified)
         (cond ((not handler) (not-found request is))
               (t (funcall handler request abs_path is)))
      (answer-request-output is request time code
                             mime-type body last-modified
                             abs_path bytes-sent))))
```

```
(defvar *document-root* "/Users/chr/Sites/"
  "Where to look for static files.")

(defclass vhost ()
  ((domainname :type 'string :accessor domainname
               :initarg :domainname
               :documentation
               "The name of the virtual host.")
   (document-root :type 'pathname
                  :accessor document-root
                  :initarg :document-root
                  :documentation
                  "Where to look for static files for the vhost.")
   (entities :type 'hashtable
              :initform (make-hash-table :test #'equal)
             :accessor entities
             :documentation
             "Dynamic resources for the virtual host.")
   (logfile :type 'pathname
            :accessor logfile
            :initarg :logfile
            :documentation
            "Location of the log file."))
  (:documentation "A virtual host."))

(defvar *vhosts* nil "List of virtual hosts")

(defmethod vhost ((hostname string))
  (find hostname *vhosts*
        :test #'string-equal
        :key 'domainname))

(defmethod vhost ((req request)) (vhost (host req)))

(defmethod vhost ((vhost vhost)) vhost)

(defmethod entities ((host string))
  (entities (vhost host)))

(defmethod document-root ((r request))
  (util::let-if (vhost (vhost r))
                (document-root vhost)
                *document-root*))

(push (make-instance 'vhost :domainname "macchr"
                     :document-root "/Users/chr/Sites/macchr/"
                      :logfile "/tmp/macchr-log")
      *vhosts*)

(push (make-instance 'vhost :domainname "localhost"
                     :document-root #p"/Users/chr/Sites/"
                      :logfile "/tmp/localhost-log")
      *vhosts*)


(defun servable-document-p (request)
  "If the file pointed to by abs_path is servable, return its entity.
 A more appropriate name is entity, I guess.
 The entity may have redirect information."
  (let ((path (pathname (uri-unescape (abs_path (uri request ))))))
    (rplaca (pathname-directory path) :relative)
    (let ((document-root (or
                           (let ((res (find (host request) *vhosts*
                                            :test #'string-equal
                                            :key 'domainname)))
                             (dbg "host: '~A', res: ~A~%"
                                 (host request) res)
                             (if res (document-root res)))
                           *document-root*)))
      (let ((entity (merge-pathnames path document-root)))
         (when (and (probe-file entity)
                    (belowp entity document-root))
           entity)))))

;; it is no longer appropriate to talk about an entity in itself, as
;; we are looking for the product of entity and method.  So, what does
;; this imply?  Is it the function we get when we look up GET entity
;; that should check whether the entity exists?  Or verify that it is
;; indeed a "virtual" entity, one that does not correspond to a
;; physical file, but appears to do so?
;;

(defparameter *options* '(OPTIONS GET HEAD POST PUT
                          DELETE TRACE CONNECT))

(defun options-document (request pathname is)
  (declare (ignore request is))
  (let ((options (remove-if-not
                  (lambda (option) (find-http-method option pathname))
                  *options*)))
    (values 200 nil "" nil
            (list "Allow"
                  (format nil "~{~A~^, ~}" options)))))


(defun ensure-relative (pathname)
  (let ((p (make-pathname :defaults pathname)))
    (rplaca (pathname-directory p) :relative)
    p))



;; a directory
;; a directory, but lacks trailing slash

;; document does not exist
;; protect against macrocharacters in pathname...

(defun serve-document  (request relative-pathname is)
  "Serve a file pointed to by PATHNAME to stream IS.
 Normally called by get and head."
  (let ((absolute-pathname
         (merge-pathnames (ensure-relative relative-pathname)
                                             (document-root request))))
    (cond ((wild-pathname-p absolute-pathname)
            (util::let-if (paths (directory absolute-pathname))
               (table-list-directory request absolute-pathname is)
             (not-found request is)))
           ((probe-file absolute-pathname)
            (if (is-directory absolute-pathname)
                (if (lacks-trailing-slash (abs_path (uri request)))
                    (moved-permanently request is
                                       (concatenate 'string
                                                    (abs_path
                                                     (uri request)) "/"))
                    (table-list-directory
                    request
                    (make-pathname :defaults absolute-pathname
                                   :name :wild) is))
                (values 200 (mime-type
                            (pathname-type absolute-pathname))
                       absolute-pathname
                        (pathname-file-last-modified absolute-pathname))))
           (t (not-found request is)))))


(defun print-file-to-output-stream (in out)
  "Read from in and write to out."
  (with-open-file (is in :element-type '(unsigned-byte 8))
    (loop with buffer-size = 2048
          with buffer = (make-array buffer-size
                                    :element-type '(unsigned-byte 8))
          for nread = (read-sequence buffer is)
          do (write-sequence buffer out :end nread)
          while (= nread buffer-size)
          ;; finally (force-output out)
          )))

(defun nwrite-sequence (sequence stream &key (start 0) (end nil))
  "Return the number of elements written to stream."
  (write-sequence sequence stream :start start :end end)
  (- (or end (length sequence)) start))
```

```
(defun zerop-safe (arg)
  "True if arg is a number and equals 0."
  (and (numberp arg) (zerop arg)))

(setf util:*dbg* nil)

(defun is-directory (entity)
  "T if entity points to a directory."
  (directoryp entity))

(defun iso-8601-string (&optional (time (get-universal-time)))
  "Return a string on the form \"2005-06-02T08:52:31\""
  (if time  (multiple-value-bind (second minute hour date
                                          month year)
                 (decode-universal-time time 0)
               (format nil "~4,’0D-~2,’0D-~2,’0DT~2,’0D:~2,’0D:~2,’0D"
                       year month date hour minute second))
      ""))

(defun list-directory
    (entity is
            &optional
            (description-format
             "~[~8< ~>~:;~:*~8,’ D~] ~36@<~A~> ~12<~A~> ~A~%"))
  "List the contents of directory ENTITY,
in the format specified by DESCRIPTION-FORMAT.
Default format suits html."
  (dolist (file (directory entity :directories t))
    (let* ((dirp (is-directory file))
            (namestring (if dirp
                            (car (last (pathname-directory file)))
                            (file-namestring-unescaped file))))
      (format is description-format
               (if (or dirp (not (probe-file file)))
                   0
                   (pathname-file-length file))
               (let ((namestring (reserved-escape namestring)))
                 (if dirp
                    (concatenate ’string namestring "/")
                    namestring))
               namestring
               (or (file-author file) "")
               (iso-8601-string (file-write-date file))))))

(defparameter *reserved-escapes*
  (let* ((reserved #.(concatenate ’string ";"  "/"  "?"  ":"
                                  "@"  "&"  "="  "+" "$"
                                  "," "\"" "#"))
          (table (make-hash-table :test ’eq :size (length reserved))))
    (loop for char across reserved
           do (setf (gethash char table)
                   (format nil "%~X" (char-code char))))
    table)
  "Characters that need escaping in URLs."
  )

(defun reserved-escape (string)
  "URL-escape string."
  (loop for char across string
         with modified
         with result = (make-array (length string)
                                  :adjustable t
                                  :element-type ’character
                                  :fill-pointer 0)
         for replacement = (gethash char *reserved-escapes*)
         if replacement
         do (progn (setf modified t)
                   (loop for replace across replacement
                         do (vector-push-extend replace result)))
         else
         do (vector-push-extend char result)
         finally (return (if modified result string))))

(defun uri-unescape (string)
  "Replace %hexhex with the corresponding character across string."
  (loop with max = (length string)
         for i below max
         with result = (make-array (length string)
                                  :adjustable t
                                  :element-type ’character
                                  :fill-pointer 0)
         with modified
         do (vector-push-extend
             (if (char= (char string i) #\%)
                 (multiple-value-bind (val pos)
                     (parse-integer string
                                    :start (min (+ i 1) max)
                                    :end (min (+ i 3) max)
                                    :radix 16
                                    :junk-allowed t)
                   (if  (< i pos)
                        (progn (setf modified t
                                     i (1- pos))
                               (code-char val))
                        #\%))
                 (char string i))
             result)
         finally (return (if modified result string))))

(defun file-namestring-unescaped (pathname)
  "The filename part of pathname."
  (if (pathname-type pathname)
      (concatenate 'string
                    (pathname-name pathname)
                    "."
                    (pathname-type pathname))
      (pathname-name pathname)))

(defun table-list-directory (request entity is)
  "List the contents of directory ENTITY as a html table."
  (let ((sink (make-string-output-stream)))
    (format sink "<html><head><title>
Directory listing of /~A</title></head>~2%<body>~%<table>~%"
             (enough-namestring (make-pathname :name ""
                                              :defaults entity)
                               *document-root*))
    (list-directory
     entity sink
     "<tr><td align=\"flight\" class=\"size\">~[~:;~:*~D~]</td>
 <td><a href=\"./~A\">~A</a></td>
 <td class=\"uname\">~A</td><td class=\"mtime\">~A</td></tr>~%")
    (format sink "</table>~%</body>~%</html>")
    (values 200 "text/html"
            (get-output-stream-string sink)
            (pathname-file-last-modified
             (make-pathname :defaults entity
                            :name nil
                            :type nil
```

```
                              :version nil)))))

;; directoryp is implementation-dependent

(defun belowp (file directory)
  "File resides hierarchically below directory."
  (let* ((path1 (pathname-directory directory))
          (mismatch (mismatch  path1 (pathname-directory file)
                               :test #'string=)))
    (or (not mismatch) (= mismatch (length path1)))))

(defun not-found-document (request)
  "The document was not found."
  (let ((uri (uri request)))
    (html-status-document
     uri
     "404 Not Found"
     "Not Found"
     (format nil
             "<p>The requested URL ~A was not found on this server.</p>"
             (abs_path uri)))))

(defun not-found (request is)
  "Document was not found."
  (declare (ignore is))
  (values 404 "text/html" (not-found-document request)))

(defun moved-permanently (request is new-abs-path)
  "Document has moved."
  (declare (ignore is))
  (values 301 "text/html"
          (moved-permanently-document request new-abs-path)))

(defun moved-permanently-document (request new-abs-path)
  "Document is moved permanently."
  (html-status-document
   (uri request)
   "301 Moved Permanently"
   "Moved Permanently"
   (format nil "<p>The document has moved <a href=\"~A\">here</A>.</p>"
           new-abs-path)))

(defun html-status-document (uri title header body)
  "Wrap body BODY in a html template."
  (format nil "<!DOCTYPE HTML PUBLIC \"-//IETF//DTD HTML 2.0//EN\">
 <html><head>
 <title>~A</title>
 </head><body>
 <h1>~A</h1>
 ~A
 <hr>
 <address>~A Server at ~A port ~A</address>
 </body></html>
 "
          title header body *server-identification*
          (host uri)
          (port uri)))

(defstruct methods
  options
  get
  head
  post
  put
  delete
  trace
  connect
  code)

(defstruct moved
  code path)

(defun find-http-method (method entity vhost)
  (let ((vhost (vhost vhost))
        (method (if (stringp method)
                    (intern (string-upcase method) #.*package*) method))
        (entity (if (pathnamep entity) entity (pathname entity))))
    (labels
        ((slot-value-safe (instance slot-name)
           (when instance
             (slot-value instance slot-name)))
         (rec (p)
           (when p
             (or (slot-value-safe (gethash p (entities vhost)) method)
                 (rec (parent-directory p))))))
      (rec entity))))
```

```
(defsetf find-http-method  set-http-method)

(defun set-http-method (method entity vhost definition)
  (let ((vhost (vhost vhost))
         (method (if (stringp method)
                     (intern (string-upcase method))
                     method)))
    (setf (slot-value (or (gethash entity (entities vhost))
                          (setf (gethash entity (entities vhost))
                                (make-methods))) method) definition)))

;; furthermore: add vhost to the calls so you get the signature
;; GET entity vhost when looking up.

(defun parent-directory (designator)
  (typecase designator
    (string (parent-directory (pathname designator)))
    (pathname (if (and (null (pathname-name designator))
                       (null (pathname-type designator))
                       (eq :unspecific (pathname-version designator)))
                  (util::let-if
                     (dir (nbutlast (pathname-directory designator)))
                     (make-pathname :directory dir
                                    :defaults designator))
                  (make-pathname :name nil
                                 :type nil
                                 :version nil
                                 :defaults designator)))))


;; add top-level handlers for each vhosts
(loop for (method handler) in '((get serve-document)
                                (head serve-document)
                                (options options-document))
      do (loop for vhost in *vhosts*
               do (setf (find-http-method method #p"/" vhost) handler)))

(export '(usage receive-request request))

(provide "http")

;; local variables:
```

```
;; eval: (modify-syntax-entry ?{  "(}  ")
;; eval: (modify-syntax-entry ?}  "){  ")
;; eval: (modify-syntax-entry ?\[ "(]  ")
;; eval: (modify-syntax-entry ?\] ")[  ")
;; eval: (put ’util::let-if ’common-lisp-indent-function 2)
;; outline-regexp: ";;;;;*"
;; indent-tabs-mode: nil
;; end:
```

## C.7 date-parser

```
(in-package http)

(defun decode-time (string)
  (values-list
   (loop for i below 3
         with index = -1
         collect (parse-integer string
                                :start (incf index)
                                :end (incf index 2)))))

(defparameter *weekdays*
  (list "Monday" "Tuesday" "Wednesday"
        "Thursday" "Friday" "Saturday" "Sunday")
  "Zero-indexed list of the days of the week,
starting with Monday.")

(defparameter *months*
  (list nil "Jan" "Feb" "Mar" "Apr"
                       "May" "Jun" "Jul" "Aug"
                       "Sep" "Oct" "Nov" "Dec")
  "One-indexed list of the months of the year,
starting with January.")

(defun indexed-format (list)
  "Return a format string that when called with a numeric argument,
produces the string found in that position in list."
  (format nil "~~[~{~@[~A~]~^~~;~}~~]" list))


(defun rfc850-date-to-universal-time (string &optional (index nil))
  (let ((index (or index (loop for i from 1
                               for day in  *weekdays*
                               for mismatch = (mismatch day string)
                               when (and mismatch
                                         (= mismatch (length day)))
                               do (return mismatch)))))
    (let* ((date (and (char= (char string index) #\,)
                      (parse-integer string
                                     :start (+ 2 index)
                                     :end (+ 2 2 index)))) ;date
           (month (position (subseq string
                                    (+ 5 index)
                                    (+ 5 3 index))
                            *months* :test #'string-equal))
           (year (parse-integer string
                                :start (+ 5 3 1 index)
                                :end (+ 5 3 1 2 index))))
      (multiple-value-bind (hour minute second)
          (decode-time (subseq string
                               (+ 5 3 1 2 1 index)
                               (+ 5 3 1 2 1 8 index)))
        (encode-universal-time second minute hour date month
                               (+ year (if (> 50 year) 2000 1900)) 0)))))

(defun rfc1123-date-to-universal-time (string)
  (let ((index 3))
    (and (char= (char string index) #\,)
         (char= (char string (incf index)) #\Space)
         (let ((date (parse-integer string
                                    :start (incf index)
                                    :end (incf index 2))))
           (and  date (char= (char string index) #\Space)
                 (util:dbg "space")
                 (let* ((month (position (subseq string (incf index)
                                                 (incf index 3))
                                         *months*
                                         :test #'string-equal))
                        (year (parse-integer string :start (incf index)
                                             :end (incf index 4))))
                   (multiple-value-bind (hour minute second)
                       (decode-time (subseq string (incf index)
                                            (incf index 8)))
                     (encode-universal-time second minute hour date
                                            month year 0))))))))


(defun asctime-date-to-universal-time (string)
  (let ((index 2))
    (and (char= (char string (incf index)) #\Space)
         (let* ((month (position
                        (subseq string (incf index) (incf index 3))
                                 *months*
                                 :test #'string-equal))
                (date (and (char= (char string (incf index)) #\Space)
                           (parse-integer string
                                          :junk-allowed t
                                          :start (incf index)
                                          :end (incf index 2)))))
           (multiple-value-bind (hour minute second)
               (decode-time (subseq string (incf index) (incf index 8)))
             (let ((year (and (char= (char string (incf index)) #\Space)
                              (parse-integer string  :start (incf index)
                                             :end (incf index 4)))))
               (encode-universal-time second minute hour date
                                      month year 0)))))))

(defun HTTP-date-to-universal-time (string)
  (let* ((index 0)
         (weekday (loop for i from 1
                        for day in  *weekdays*
                        for mismatch = (mismatch day string)
                        when (and mismatch (= mismatch (length day)))
                        do (progn (setf index mismatch) (return i)))))
    (if weekday                                ;rfc850-date
        (rfc850-date-to-universal-time string index)
                                        ;else rfc1123 or asctime
        (if (char= (char string 3) #\,)        ;rfc1123-date has a comma
            (rfc1123-date-to-universal-time string)
            (asctime-date-to-universal-time string)))))

(provide ’date-parser)
```

## C.8 md5

```
(defpackage :md5 (:use "COMMON-LISP"))

(in-package md5)

(eval-when (:compile-toplevel :load-toplevel :execute)

  (defparameter *t-table*
    ;; in my cl,
    ;; (floor (* 4294967296 (abs (sin 1))))
    ;; -> "#xd76aa400", therefore I use the values from the spec.
    ;; the zero'th element is not used
    #(#x00000000 #xd76aa478 #xe8c7b756 #x242070db #xc1bdceee #xf57c0faf
      #x4787c62a #xa8304613 #xfd469501 #x698098d8 #x8b44f7af #xffff5bb1
      #x895cd7be #x6b901122 #xfd987193 #xa679438e #x49b40821 #xf61e2562
      #xc040b340 #x265e5a51 #xe9b6c7aa #xd62f105d #x02441453 #xd8a1e681
      #xe7d3fbc8 #x21e1cde6 #xc33707d6 #xf4d50d87 #x455a14ed #xa9e3e905
      #xfcefa3f8 #x676f02d9 #x8d2a4c8a #xfffa3942 #x8771f681 #x6d9d6122
      #xfde5380c #xa4beea44 #x4bdecfa9 #xf6bb4b60 #xbebfbc70 #x289b7ec6
      #xeaa127fa #xd4ef3085 #x04881d05 #xd9d4d039 #xe6db99e5 #x1fa27cf8
      #xc4ac5665 #xf4292244 #x432aff97 #xab9423a7 #xfc93a039 #x655b59c3
      #x8f0ccc92 #xffeff47d #x85845dd1 #x6fa87e4f #xfe2ce6e0 #xa3014314
      #x4e0811a1 #xf7537e82 #xbd3af235 #x2ad7d2bb #xeb86d391))

  (defparameter *old-readtable* (copy-readtable))

  (setf *readtable* (copy-readtable nil))

  (set-macro-character
   #\[
   #'(lambda (stream char1)
       (declare (ignore char1))
       (destructuring-bind (a b c d k s i)
           (append (loop for i below 4
                         collect (intern
                                  (string
                                   (read-char stream nil nil t))))
                   (list* (read stream nil nil)
                          (read stream nil nil)
                          (read stream nil nil)
                          (read-delimited-list #\] stream t)))
         `(setf ,a (word (+ ,b
                          (<<< (word (+ ,a
                                        (funcall log-op ,b ,c ,d)
                                        (aref x ,k)
                                        ,(svref *t-table* i)))
                               ,s)))))))

  (set-macro-character #\] (get-macro-character #\))))

(defun md5 (message)
  (let ((vector (make-array 16 :fill-pointer 0))
        (sum 0)
        (weight 0)
        (nbytes 0)
        (a #x67452301)
        (b #xefcdab89)
        (c #x98badcfe)
        (d #x10325476))
    (labels ((word (integer) (logand #xFFFFFFFF integer))
             (<<< (integer count)
               (logior (ash (logand
                             (byte (- 32 count) 0)
                             integer) count)
                       (ash (logand (byte count (- 32 count))
                                    integer)
                            (- count 32))))
             (f (x y z) (logior (logand x y) (logandc1 x z)))
             (g (x y z) (f z x y))
             (h (x y z) (logxor x y z))
             (i (x y z) (logxor y (logorc2 x z)))
             (operations (a b c d x)
(let ((log-op #'f))
  [ABCD  0  7  1] [DABC  1 12  2] [CDAB  2 17  3] [BCDA  3 22  4]
  [ABCD  4  7  5] [DABC  5 12  6] [CDAB  6 17  7] [BCDA  7 22  8]
  [ABCD  8  7  9] [DABC  9 12 10] [CDAB 10 17 11] [BCDA 11 22 12]
  [ABCD 12  7 13] [DABC 13 12 14] [CDAB 14 17 15] [BCDA 15 22 16])
(let ((log-op #'g))
  [ABCD  1  5 17] [DABC  6  9 18] [CDAB 11 14 19] [BCDA  0 20 20]
  [ABCD  5  5 21] [DABC 10  9 22] [CDAB 15 14 23] [BCDA  4 20 24]
  [ABCD  9  5 25] [DABC 14  9 26] [CDAB  3 14 27] [BCDA  8 20 28]
  [ABCD 13  5 29] [DABC  2  9 30] [CDAB  7 14 31] [BCDA 12 20 32])
(let ((log-op #'h))
  [ABCD  5  4 33] [DABC  8 11 34] [CDAB 11 16 35] [BCDA 14 23 36]
  [ABCD  1  4 37] [DABC  4 11 38] [CDAB  7 16 39] [BCDA 10 23 40]
  [ABCD 13  4 41] [DABC  0 11 42] [CDAB  3 16 43] [BCDA  6 23 44]
  [ABCD  9  4 45] [DABC 12 11 46] [CDAB 15 16 47] [BCDA  2 23 48])
(let ((log-op #'i))
  [ABCD  0  6 49] [DABC  7 10 50] [CDAB 14 15 51] [BCDA  5 21 52]
  [ABCD 12  6 53] [DABC  3 10 54] [CDAB 10 15 55] [BCDA  1 21 56]
  [ABCD  8  6 57] [DABC 15 10 58] [CDAB  6 15 59] [BCDA 13 21 60]
  [ABCD  4  6 61] [DABC 11 10 62] [CDAB  2 15 63] [BCDA  9 21 64])
               (values a b c d))
             (dump-byte (byte)
               (if (eq byte :eof)
                   ;; value of nbytes before padding
                   (let ((bitlength (* 8 nbytes))
                         (remainder (mod nbytes 64)))
                     (dump-byte 128)
                     (dotimes (i (- (if (< remainder 56) 56 120)
                                    remainder 1))
                       (dump-byte 0))
                     (loop for i below 64 by 8
                           do (dump-byte (ldb (byte 8 i) bitlength)))
                     (setf sum 0
                           (fill-pointer vector) 0
                           weight 0
                           nbytes 0)
                     (format nil "~{~2,'0X~}"
                             (mapcan
                              (lambda (num)
                                (mapcar (lambda (i)
                                          (ldb (byte 8 i) num))
                                        (list 0 8 16 24)))
                              (list a b c d))))
                   (progn
                     (setf sum (dpb byte (byte 8 weight) sum))
                     (incf nbytes)
                     (when (= (incf weight 8) 32) ; logand
                       (vector-push sum vector)
                       (setf sum 0
                             weight 0))
                     ;; >= integer 16 == (logbitp 4 integer)
                     (when (= (fill-pointer vector) 16)
                       ;; as opposed to the spec, aa bb cc dd
                       ;; are the new values for a b c d
                       (multiple-value-bind (aa bb cc dd)
                           (operations a b c d vector)
                         (setf a (+ a aa)
                               b (+ b bb)
                               c (+ c cc)
                               d (+ d dd)
                               (fill-pointer vector) 0)))))))
      (declare (inline word <<< f g h i))
      (cond
        ((stringp message)
         (map nil (util::compose #'dump-byte #'char-code) message))
        ((and (streamp message) (input-stream-p message))
         (if (subtypep (stream-element-type message) 'character)
             (loop for element = (read-char message nil :eof)
                   until (eq element :eof)
                   do (dump-byte (char-code element)))
             (loop for element = (read-byte message nil :eof)
                   until (eq element :eof)
                   do (dump-byte element))))
        ((pathnamep message) (with-open-file
                                 (in message :direction :input)
                               (return-from md5 (md5 in)))))
      (dump-byte :eof))))

(eval-when (:compile-toplevel :load-toplevel :execute)
  (setf *readtable* *old-readtable*))

(defun md5test ()
  (loop for (message correct) in
        '(("" "d41d8cd98f00b204e9800998ecf8427e")
          ("a" "0cc175b9c0f1b6a831c399e269772661")
          ("abc" "900150983cd24fb0d6963f7d28e17f72")
          ("message digest" "f96b697d7cb7938d525a2f31aaf161d0")
          ("abcdefghijklmnopqrstuvwxyz"
           "c3fcd3d76192e4007dfb496cca67e13b")
          (#.(concatenate 'string "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
              "abcdefghijklmnopqrstuvwxyz0123456789")
           "d174ab98d277d9f5a5611c2c9f419d9f")
          (#.(concatenate 'string
              "123456789012345678901234567890"
              "1234567890123456789"
              "0123456789012345678901234567890")
           "57edf4a22be3c955ac49da2e2107b67a"))
        collect (format nil "~S~%~S~%~(~S~)~2%"
                        message correct (md5 message))))
```

## C.9 server-process

```
(defpackage :server (:use  :http :meta :common-lisp))

(in-package :server)

#+lispworks (require "lispworks-server-process"
                     "compat/lispworks/server-process.lisp")
#+openmcl (require "openmcl-server-process"
                   "compat/openmcl/server-process.lisp")

(eval-when (:compile-toplevel :load-toplevel)
  (require "http" '("http")))

(defun start ()
  "Start a server."
  (let ((proc (find "server" (process-list)
                    :test #'string=
                    :key #'process-name)))
    (when proc (process-kill proc)))
  (load (make-pathname  :name server::*configuration-name*
                        :type server::*configuration-type*
                        :defaults server::*configuration-directory*))
  (start-up-server :function 'run-server
                   :announce nil
                   :service 8000
                   :process-name "server"))

(defun stop ()
  "Stop the server"
  (loop for proc = (find "slave" (process-list)
                         :test #'string=
                         :key #'process-name)
        while proc do (process-kill proc))
  (let ((proc (find "server" (process-list)
                    :test #'string=
                    :key #'process-name)))
    (when proc (process-kill proc))))

(defun run-server (s)
  "Listen to a socket S."
  (unwind-protect
          ;; you may want to name the thread with the remote-host?
          ;; reply with 503 Service Unavailable if there is too much work...
      (loop (process-run-function "slave"  #'accept-and-receive
                                  (accept-connection s :wait t))
          (util:dbg "started a slave.~%"))
    (close s)))

(defun accept-and-receive (is)
  "Handle requests on the input stream IS."
  (when is (unwind-protect
               (do ((result (http:receive-request is)
                            (http:receive-request is)))
                   ((not result)))
             (close is))))

(defvar *configuration-directory*
  #p"/Users/chr/proj/uio/sli/hovedfag/*.*"
  "Where to look for the configuration file.")
(defvar *configuration-name* "config"
  "Name, sans suffix, of the configuration file.")
(defvar *configuration-type*
  "lisp" "Suffix of the configuration file.")

(export '(start stop))
(provide "server-process")
```

## C.10 clack.asd

```
(defsystem "clack"
    :description "clack: a web server."
    :version "1.0"
    :author "Christian Nybø <chr@sli.uio.no>"
    :licence "Public Domain"
    :components ((:file "utilities")
        (:file "meta" :depends-on ("utilities"))
        (:file "bnf" :depends-on ("meta"))
        (:file "http" :depends-on ("bnf"))
        (:file "server-process" :depends-on ("http"))))
```

## C.11 sample-rules.txt

Some of the rules are reformulated, then the original is commented above the reformed rule.

As latex has trouble with a file containing characters from latin-1 such as guillemot and textcent at the same time, I have provided but a subset of this file.

```
          SP           = " "
          HT           = "^I"
          LF           = "
"
          CR           = "^M"
          HTTP-date    = rfc1123-date | rfc850-date | asctime-date
          wkday        = "Mon" | "Tue" | "Wed"
                       | "Thu" | "Fri" | "Sat" | "Sun"
          http_URL = "http:" "//" RFC2396::host [ ":" RFC2396::port ]
                     [ RFC2396::abs_path [ "?" RFC2396::query ]]
          rfc1123-date = wkday "," SP date1 SP time SP "GMT"
          rfc850-date  = weekday "," SP date2 SP time SP "GMT"
          asctime-date = wkday SP date3 SP time SP 4DIGIT
          date1        = 2DIGIT SP month SP 4DIGIT
                         ; day month year (e.g., 02 Jun 1982)
          date2        = 2DIGIT "-" month "-" 2DIGIT
                         ; day-month-year (e.g., 02-Jun-82)
          date3        = month SP ( 2DIGIT | ( SP 1DIGIT ))
                         ; month day (e.g., Jun  2)
          time         = 2DIGIT ":" 2DIGIT ":" 2DIGIT
                         ; 00:00:00 - 23:59:59
          weekday      = "Monday" | "Tuesday" | "Wednesday"
                       | "Thursday" | "Friday" | "Saturday" | "Sunday"
          month        = "Jan" | "Feb" | "Mar" | "Apr"
                       | "May" | "Jun" | "Jul" | "Aug"
                       | "Sep" | "Oct" | "Nov" | "Dec"
          delta-seconds  = 1*DIGIT
          charset = token
          content-coding   = token
          transfer-coding         = "chunked" | transfer-extension
          transfer-extension      = token *( ";" parameter )
          parameter               = attribute "=" value
          attribute               = token
          value                   = token | quoted-string
          Accept         = "Accept" ":"
```

```
                         #( media-range [ accept-params ] )
;          media-range    = ( "*/*"
          media-range    = *LWS
                         ( "*/*"
                         | ( type "/" "*" )
                         | ( type "/" subtype )
                         ) *( ";" parameter )
          accept-params  = ";" "q" "=" qvalue *( accept-extension )
          accept-extension = ";" token [ "=" ( token | quoted-string ) ]
;          comment        = "(" *( ctext | quoted-pair | comment ) ")"
          comment        = (*LWS "(" *( ctext | quoted-pair | comment ) ")")
                          | ("(" *( ctext | quoted-pair | comment ) ")")
```

## C.12 mime-filter

```
mime-list = (comment  | type-suffixes | type | blank) newline
newline = "
"
comment = "#" *NOT-NEWLINE
blank = *" "
type-suffixes = type (" "|"        ") *(" "|"        ") suffixes
type = token "/" token
suffixes = suffix *suffix
suffix = token
token = TOKEN-CHAR *TOKEN-CHAR
```