A persistent web application for distribution of weather information in Scheme

Mario Latendresse

Northrop Grumman
Fleet Numerical Oceanography and Meteorology Center (FNMOC)
Navy Research Laboratory
Monterey, CA

Abstract. For several years, the Navy has used Scheme to build a web application for the distribution of real-time weather information for mission plannings. This application, called Metcast, is composed of meteorological bulletin decoders and a web server accessible around the world. The decoders populate several databases of real-time weather observation reports, forecasts, satellite images and gridded data produced by weather models. The requests to Metcast are formulated in a simple language, its syntax being s-expressions. The responses are returned as either binary data or XML.
Recently the Metcast server was modified to run under the fastCGI module of Apache. This module maintains a pool of persistent web application instances, each capable of answering multiple requests. This persistency increases throughput by avoiding multiple process creations, startup delay and reconnection to database servers.
A new tool is being used to generate lexical parsers in Scheme. It is based on a cascade of lexers, described by tagged regular expressions (TREs) written as s-expressions. In Metcast, these lexical parsers are used to generate meteorological decoders. We advocate TREs, instead of Perl style regular expressions, as they can do complex lexical parsings by generating parse trees which Lisp like languages can easily manipulate.
We present the latest Metcast architecture and the technical details of the integration of Scheme with other fundamental Unix tools. We argue that the Metcast implementation, using Scheme, is highly portable and could not have been accepted by the Navy if it were not for its full integration with common Unix tools. We also pinpoint some basic functionalities unavailable in the Scheme standard (R5RS) but for which our web application could not function properly and efficiently.

1 Introduction

The Fleet Numerical Meteorology and Oceanography Center (FNMOC), in Monterey, has the mandate to distribute real-time weather information for the Navy mission plannings. The Metcast system, originally created by Oleg Kiselyov [8], has been successfully used for several years to distribute this information as a web application. It is composed of a set of decoders populating databases, and

a web application – the Metcast server – answering requests formulated as s-expressions. The responses are dynamically generated based on database tables. The Metcast server currently answers over twenty thousands HTTP requests a day. The response sizes range from a few bytes to tens of millions of bytes, the average being around 500 KB.

In its original design, the Metcast server uses CGI, but each request spawns five processes, reaching the processing power limit of a Sun E10K[1].

Metcast has recently been adapted to run under the fastCGI [1] module of Apache in order to reduce this load. This new implementation dramatically reduces the number of spawned processes. Moreover, it will allow a distributed architecture on clusters of computers as new hardware will replace the multiprocessor computer.

A proxy, written in C, uses the API provided by the fastCGI development kit [1]. In essence, it is the bridge between the fastCGI module and the Scheme code. It transforms an HTTP request into a syntax easily readable by the Scheme code.

Several proxy instances may be running in parallel. They are spawned and killed by the fastCGI module. The behavior of the spawning algorithm is configurable by a user defined directive in file `httpd.conf`.

A spawned proxy initiates a connection on a Unix port for which the Unix superdaemon `inetd` has been configured to monitor. Such a connection initiates, by `inetd`, a new Metcast server instance which communicates with the proxy through a TCP connection. The Metcast instance can be located on a different computer. All HTTP requests go through the fastCGI module, a proxy instance and the corresponding Metcast server. All responses go through the reverse path.

The databases are populated by a set of decoders permanently running. WMO (World Meteorological Organisation) bulletins are bundled into large files dropped into a specific directory. They are pickup by the decoders, decoded and ingested into a database. Similarly, other data, like satellite imageries and gridded data produced by forecasting models are ingested into databases.

Section 2 presents the MBL request language for Metcast. In section 3 we present the technical details to adapt some Scheme code to run under fastCGI as a dynamic web application. In section 4 we discuss the necessary exception handling features a Scheme implementation should provide to implement a resilient web application. The database connection technique used is presented in section 5. Section 6 presents a new tool to construct complex lexers using s-expressions as a descriptive language. We use these to build WMO bulletin decoders. In section 7 we compare our approach to some similar works. Finally, we conclude in section 8.

[1] The E10K is a ten processors Sun computer also used to run all the decoders and the database servers.

```
(example1
  (bounding-box 50 -77 42 -20)
  (st_constraint (st_country_code "CN"))
  (products (METAR TAF)))
```

Fig. 1. An MBL request for TAF and METAR products.

2 The Metcast request language

As most web applications, two kinds of requests can be made to Metcast: post and get. A get request, for which the URL itself specifies the parameters, provides a simple mechanism to obtain the catalog of available weather products. A post request allows a message content. We use it to pass an s-expression called an MBL expression. It is a list formed by global parameters and clauses identifying the requested weather products.

Fig. 1 presents a simple MBL request: example1 is the name of the request, the region is specified by the bounding-box with latitudes and longitudes, a global constraint specifying a country, and the product list: METAR and TAF. The result of this request is a multi-part HTTP response formed by two OMF (XML instance) documents.

Since an MBL expression is a native data structure for Scheme, it can be read by the primitive `read`. This primitive is actually too simple as requests could be badly formed. This could hang the reader or raise a run-time error. For example, the s-expression '`(a (products TAF)`' lacks a closing parenthesis causing `read` to wait forever on the TCP stream. Assuming that run-time errors can be caught, a more robust solution is to read the content of a post request as a string, given the content length of the post request, and use this string as a port for `read`. This is the adopted technique in Metcast.

This is an important reason to be able to catch any Scheme internal exceptions, a non-standard feature in many Scheme implementations.

3 Adapting a Scheme application to run under fastCGI

Web applications using CGI suffer from starting time due to process creations, code loading and database connections. The fastCGI module under the Apache server tries to diminish this shortcoming by providing dynamic persistent web applications. The module maintains a pool of running applications capable of answering several requests. It spawns and kill these processes according to the number of requests and responses delay.

Running a dynamic web application under the fastCGI module entails that all input/output must follow a specific protocol. Technically, a dynamic fastCGI application must communicate through the fastCGI API provided by the development kit [1]. To shield the Scheme code from any direct interaction with this API, and still use standard input/output, we wrote a proxy in C. The proxy uses

this API and communicates with the Scheme code through a TCP connection, although the Scheme code is relatively unaware of this. We use the superdaemon `inetd` which listens to a port and spawns a new Metcast server instance when a proxy initiates a communication. The Metcast server does all input/output using the standard primitives since `inetd` takes care of the TCP connection.

The greatest complication is that the end of file indicator can no longer be used. So, all outputs are translated into strings and sent in chunks to the proxy, with explicit lengths. An explicit zero byte chunk indicates the end of file. The input is simpler, as we only have to read the body content of a post request.

Metcast relies on two programs to retrieve images and gridded data from a database. The resulting responses for this type of data can be large, often greater than 5 MB. Since their output already uses the chunk encoding, we simply relay this data, using a port-copy, to the proxy, without going through the Scheme code. It greatly increases performance.

The Metcast system currently uses Gambit [5] (version 3.0), a Scheme implementation built by Marc Feeley. We had to write several foreign functions (in C) to implement low level Posix system calls. In particular, some undocumented features of Gambit had to be used to catch run-time errors and gracefully end the Metcast server.

We have also considered other implementations, in particular Bigloo developed by Manuel Serrano [10]. Some of its particularities would reduce programming web applications: internal exceptions can be caught, and some of the Posix system calls are already implemented.

4 Metcast resiliency

It is important that the Metcast system be resilient and gracefully report programming errors back to the proxy. Scheme is a dynamically typed language and several programming errors are reported at run-time. We often add new features and modules, and debugging run-time errors can be difficult if the Scheme implementation ends without reporting back any error messages. This is what would happen if we let it exit and write to standard output any error messages, since the proxy expect only chunk encoded output.

For the Scheme implementations considered, a run-time error generates an internal exception signal. For the server it is essential that these exceptions be caught such that an 'HTTP 500' error be properly reported back to the client through the proxy. Moreover, the Metcast server may find itself in a state where it can no longer accept any requests: it must then exit and signal the proxy to shutdown. Otherwise, the proxy would accept a new request and try to serve it, although the Metcast instance would have exited for good.

For the decoders, the general strategy is to run the main decoder module through a shell script. It should run indefinitely. If it exits, an run-time error occured. The shell script restarts it, and if this rapidly happens two times, the current file of WMO bulletins is moved away into an exception directory. In this manner, if the error is due to a new decoder module, all the files containing

the corresponding bulletins are skipped, yet preserved for latter decoding. Obviously, a corrective action should soon be done, but all other bulletins are usually processed keeping the whole Metcast system alive.

5 The database connection

We use an Informix database server. The communication between the Scheme code and this database server is done through `dbaccess`. Actually, this tool is mainly designed for interactive communication with the server. Yet, it can easily be used through a bidirectional named pipe. An MBL request is translated into a SQL request, which is transfered to the server through `dbaccess`. The response is read through the pipe and formatted as a XML document for text or transfered as is for binary data[2].

To increase the robustness of the Metcast server, all database server communications are wrapped by a timer mechanism that raises a signal after a specified amount of time.

The simplicity of the database connection requires only two special features from a Scheme implementation: the creation of a bidirectional named pipe and a timer mechanism. Despite this simplicity, it is robust, portable and efficient.

6 RegReg: Generation of complex lexical parsers

The Metcast server emits weather observations and forecasts as XML documents. Since these documents have a well defined structure, with tags, they can be easily manipulated using languages and tools such as SLT, XPATH, etc. This is an essential feature of Metcast: it converts complex untagged WMO bulletins into well tagged XML documents.

The ability to emit such XML comes from the well structured database tables which have been populated by WMO meteorological bulletins. These bulletins have a complex and terse lexical format defined in the 1950s by WMO. In many cases they are human generated. It takes a trained meteorologist to reliably encode and decode them.

To give a feeling of the lexical nature of the WMO bulletins, Figures 2 and 3 present an AIRMET and TAF (Terminal Aerodrome Forecast) bulletins respectively.

The first line of the AIRMET bulletin gives the general area 'BOS', 'Z' being the type, a 'ZULU' report, '201014' being the issue-time of the report in the format day, hour and minute. The second line provides the update number, the specific reason for this report, 'ICE', and the time this report becomes obsolete. On the first line of the second paragraph the affected regions are specified as 'PA OH WV ... WTRS'. The second line is more specific with a series of relative airport locations specifying a closed polygon, each location being an airport name and an optional distance and direction; for example, '60NW CYN' means

[2] Satellite images and gridded data are in binary format; all WMO bulletins are text.

```
BOSZ WA 201014
AIRMET ZULU UPDT 4 FOR ICE AND FRZLVL VALID UNTIL 201200
.
AIRMET ICE...PA OH WV VA MD DC DE NJ AND CSTL WTRS
FROM AIR TO 60NW CYN TO 30NEE ORF TO HMV TO HNN TO AIR
LGT OCNL MOD RIME OR MXD ICGICIP BTN 140 AND FL250. CONDS CONTG
BYD 02Z SPRDG EWD THRU 08Z.
.
OTLK VALID 0200-0800Z...ICE NJ DE MD VA CSTL WTRS
S LN CYN-75S ACK AND N LN ORF-150ESE SBY LGT OCNL MOD RIME OR MXD
ICGICIP ABV 140 TO FL250. CONDS CONTG THRU 08Z.
.
FRZLVL...100 NR AND N LN YQB-PQI
         120 NR LN YOW-CON-140SE BGR
         140 NR LN ASP-ERI-APE-BKW-SIE-140E ACK
         140-145 RMNDR AREA
....
```

Fig. 2. An AIRMET bulletin.

```
TAF KCTB 072330Z 080024 26019G26KT P6SM SCT100 BKN150
FM0200 29014KT P6SM SC T100 BKN150 FM1000 35013KT P6SM SCT050
BKN100 PROB30 1216 3SM -SN BR BKN040 FM1600 34017G26KT P6SM
SCT070 BKN120=
```

Fig. 3. A TAF bulletin.

60 miles North West of CYN. The third line described in more detail the icing observation with a specification of a height range 'BTN 140 AND FL250'; this last term being in fact 25000 feet.

The TAF bulletin shows even more examples of tokens from which several parts must be extracted to find out its meaning. The token '34017G26KT' specifies wind of 17 knots, 340 degrees from the North, gusting at 26 knots; the gusting part is optional as in '29014KT'.

There is a decoder for each type of WMO bulletins: TAF, METAR, SYNOPS, SIGMETS, AIRMETS, etc. Currently we have twelve decoders but we still have many to write and these current ones must be maintained to increase their reliability[3]. All decoders were written in Scheme.

Due to the complexity of building and maintaining them it was decided to write a general tool to emit lexical parsers, or lexers, based on a description of the WMO bulletins.

There are tools capable of generating lexers in Scheme from regular expressions (e.g. SILex [3]). These tools are not adapted for stand-alone syntax analysis,

[3] It is often discovered that meteorologists do not input bulletins exactly as officially specified. Once these discrepencies are discovered we modify the decoders.

```
a) ((? (= d "[0-3][0-9]")) (= h "[0-2][0-9]") (= m "[0-5][0-9]") (? "Z"))

b) ((* "0") (= x ("[1-9]" (* "[0-9]"))))
```

Fig. 4. Two tagged regular expressions (TREs) with bindings.

but rather for compiler construction. They work mainly as recognizer, a major drawback to be used as parser. In other words, they do not return parts of the recognized strings, simply the whole result.

Regular expressions can describe WMO messages, but they lack the possibility of easily extracting parts of the recognized strings. As discussed, for WMO bulletins, tokens must often be broken in several parts to extract their meaning.

All known lexical generators, libraries and languages based on regular expressions (e.g. Perl) have poor capabilities for extracting parts of the recognized strings. It is possible to partially extract some parts, but it is not for many practical situations. For example, in Perl, the regular expression `/([0-9])G([0-9])/` creates two bindings, for the parentheses subexpressions, namely `$1` and `$2`. These are bound to the substrings matching the first and second subexpression `[0-9]` respectively. For the string `4G7` it would binds `4` to `$1` and `7` to `$2`. But the expression `/(([0-9])G([0-9]))+/` is problematic since `$1` is not necessarily bound to the first matching of `[0-9]`, it could be bound to the second or the entire matching of `([0-9])G([0-9])`. The mechanism of subexpression matching was not designed for complex cases, but only for simple non-nested subexpressions.

Longterm efficiency should also be a concern: the generated code should use an efficient technique. Perl regular expression implementation is based on nondeterministic automaton, that is it uses backtracking. Our tool is based on a deterministic technique, requiring no backtracking and therefore avoiding the potential exponential time execution.

Moreover, instead of using two separate tools, as for example Lex and Yacc, the lexer **and** parser are described by regular expressions; actually tagged regular expressions.

Fig. 4a shows a tagged regular expression (TRE) representing a timestamp. The operator `=` binds the tags d, h and m with the corresponding parts of the token. The operator `?` specifies an optional part; an expression as `"[0-9]"` represents the set of characters '0' to '9'. Assuming that the string to parse is `"311230"`, the resulting parse tree would be `((d "31") (h "12") (m "30") ())`; if '31' were missing, the result would be `(() (h "12") (m "30") ())`. In general, the value of a tag is the subtree associated with it in the parse tree. The resulting tree is easy to manipulate as one general function `tag->value` returns the value of a given tag in a parse tree; if the tag does not exist, it returns `#f` (false). Fig. 4b is a TRE for positive integers where x would be bound to the significant part.

```
(regreg
  (rules 1
    (wind    ((= speed (+ "[0-9]")) (? "G" (= gust (+ "[0-9]"))) "KT")
    (time    ((+ "[0-9]") "Z"))
    (area    "[A-Z][A-Z][A-Z]")
    (outlook (or "ICG" "TURB" "MTN OBSC" "LLWS"))
    (dir     ((= degree (+ "[0-9]")) (or "E" "N" "W" "S")))
    (b       (+ " "))
    (end     "=")))

  (rules 2
    (msgA    (time (+ b area b wind b dir) b end))
    (msgB    ((= period (time b time)) b area b outlook b end)))

  (rules 3
    (bulletin  (+ (or msgA msgB)) process-bulletin)))
```

Fig. 5. A small decoder specification using TREs.

As in Lex, the set of valid tokens is described by a series of TREs. The parser is also described by TREs, actually a cascading series of TREs grouped in rules sections.

A TREs such as `(* (= x "[A-Z]") "[0-9]")`, where '*' is the kleene closure, may create several bindings for the same tag; parsing the string `"A1B2C3"` would result into the parse tree `((x "A") "1" (x "A") "2" (x "A") "3")`. This presents no difficulty as the semantic analysis expect such multiple bindings and should act accordingly. One general function `tag->all-values` — that returns the list of all values, given a tag, by inorder traversal of the parse tree — suffices for extracting the multiple bindings.

Parsing ambiguities can arise in some specifications. For example, the TRE `((? (= x "[0-9]")) (? (= y "[0-9]")))` is ambiguous if the string `"3"` is parsed since '3' can be validly bound to x or y.

Fig. 5 describes a small parser using three rules sections. The first section describes the basic tokens based on characters. In compiler term, this is the lexer. The second section is based on the first, the lexer, while the third is based on the second. These two sections form the parser. The same TRE semantic apply to all sections: TREs are matched using the longest match, unless the option `short` is specified; when two TRE matches, the TRE occuring sooner in a section has precedence. In rules sections 2 and 3, the basic literals are no longer characters but tokens although the syntax is described by TREs. The advantage over a full LALR(1) grammar is conciness: no basic operations have to be specified to build a tagged parse tree. The tree itself always have the same basic form, making simple their manipulation.

In rules section 3, when a bulletin has been recognized, `process-bulletin` is called with the constructed parse tree as argument. Each rule can be associated

with a function. An error keyword can be specified in each section to name a function to call when the parser cannot match the input.

A tool written in Scheme – called RegReg – generates a lexer table based on a cascade of rules sections. The table is combined with a driver to form a complete lexer/parser. The implementation technique used by RegReg is an extention of the technique presented by Dubé [4]. It essentially associate list operations with NFA arcs. The transformation from NFA to DFA creates chains of operations. We have modified the method by adding an operation to bind tags to subtrees.

The fact that Scheme can natively read and manipulate complex data structures, similar to XML, is important to promote it as a general language to build specialized tools. We believe that Scheme strong point is its minimalist general and powerful core upon which more specialized tools and languages can be built in a short period of time. These have been important facts to promote Scheme at FNMOC.

7 Other approaches for Scheme web applications

PLT Scheme (aka DrScheme)[6] libraries include a web server [7] and a framework to program web applications. It is an elegant solution but the main drawback of this approach is the necessity to adopt an all out Scheme solution. It is not aimed at reusing well established servers, like Apache. It is also not well adapted for clusters as all servlets are running in the same interpreter instance.

Ilia Perminov wrote a PLT extension to run Scheme code under fastCGI of Apache [9]. This approach does not use the dynamic load balancing facility of fastCGI; and this cannot be used to scale the web application to a cluster of computers. Furthermore, the input/output uses TCP connection explicitly, requiring new primitives to open, close, read and write; whereas our implementation directly uses the standard input/output leaving this work to the Unix superdaemon `inetd`.

The `mod_pipe` module of Anikin and Lisovsky [2] ressembles our approach as it does not depend on any Scheme implementation; but it is aimed at dynamically transforming web page contents in a similar manner to the PHP language.

8 Conclusion and future work

Web applications written in Scheme can be well integrated with existing web servers, even with the latest facilities as fastCGI. Our approach relies on common tools and basic functionalities of an operating systems (e.g. superdaemon `inetd`, named pipes, etc.).

Integration with well known web servers and their added modules is a very important feature for industrial acceptance. An all out Lisp approach would not have be accepted by the U.S. Navy.

The Srfi (Scheme Request For Implementation) process has introduced new helpful libraries for web applications, but more needs to be done. In particular,

Posix like system calls are a necessity to better interact with the operating system. Probably, a Scheme srfi should be proposed to deal with web applications needs.

Several Scheme implementations could have been used to meet our requirements, but, to that end, some are better than others. The ideal implementation would have Posix system calls, simple internal exception handlings, and timer mechanism.

We believe that Perl like regular expressions are an ill adapted approach for Lisp like languages. We believe it is more appropriate to generate tagged parse trees, from tagged regular expressions as in RegReg, which are easily manipulable as s-expressions.

Finally, s-expressions are not industrially well known, but the XML acceptance by non-academic institutions makes s-expressions and the family of Lisp languages more acceptable.

9 Acknowledgments

Many thanks to: Oleg Kiselyov for many technical tips in adapting Metcast under fastCGI; Marc Feeley for Gambit; Manuel Serrano for some discussion on his Bigloo system; and Danny Dubé for his prompt replies about his paper.

References

1. fastcgi home page, [August 2002]. http://www.fastcgi.com/.
2. Victor Anikin and Kirill Lisovsky. mode_pipe, [August 2002]. http://www196.pair.com/lisovsky/mod_pipe/.
3. Danny Dubé. SILex, [August 2002]. http://www.iro.umontreal.ca/~dube/.
4. Danny Dubé and Marc Feeley. Efficiently building a parse tree from a regular expression. *Acta Informatica*, 37(2):121–144, 2000.
5. Marc Feeley. Gambit Scheme System, [August 2002]. http://www.iro.umontreal.ca/~gambit/.
6. Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, and Shriram Kristnamurthi. The drscheme project: An overview. *SIGPLAN Notices*, 33(6):17–23, 1998.
7. Paul Graunke, Shriram Krishnamurthi, Steve Van Der Hoeven, and Matthias Felleisen. Programming the Web with high-level programming languages. *Lecture Notes in Computer Science*, 2028:122–135, 2001.
8. O. Kiselyov. Implementing Metcast in Scheme. In Matthias Felleisen, editor, Proceedings of the Workshop on Scheme and Functional Programming, volume Rice COMP TR00-368, pages 23–25, Montreal (Canada), September 2000.
9. Ilia Perminov. Plt scheme libraries and extensions, [August 2002]. http://www.cs.utah.edu/plt/develop/.
10. Manuel Serrano. Bigloo Home Page, [August 2002]. http://www-sop.inria.fr/mimosa/fp/Bigloo/.