Implementing Metcast in Scheme*

Oleg Kiselyov
Software Engineering, Naval Postgraduate School, Monterey, CA 93943
oleg@pobox.com, oleg@acm.org

Abstract

This paper presents a case study of implementing a large distributed system in Scheme. *Metcast* is a request-reply and subscription system for dissemination of real-time weather information. The system stores a large amount of weather observation reports, forecasts, gridded data produced by weather models, and satellite imagery. A Metcast server delivers a subset of these data in response to a query formulated in a domain-specific language. Decoders of World Meteorological Organization's data feed, the Metcast server, XML encoders and decoders, auxiliary and monitoring CGI scripts are all written in Scheme.

This paper considers two examples that demonstrate benefits of our choice of the implementation language: parsing of the data feed and a module system for the Metcast server. We will also discuss extensions to Scheme as well as performance.

1 Overview of Metcast

Metcast is a request-reply and a subscription system for distributing, disseminating, publishing and broadcasting of real-time weather information [1]. The system comprises clients and servers communicating in an HTTP protocol. A Metcast server maintains a database of weather observation reports, forecasts, advisories, gridded data produced by weather models, as well as of satellite imagery and plain text messages and discussions. A Metcast client uses a web form or a domain-specific, flexible request language to retrieve a subset of data from a Metcast database [2]. A Metcast server – which is an application (web) server – parses requests, queries the database and sends the requested data in a single- or a multi-part reply. A server may act as a client to request a subset of data for further redistribution. Metcast servers are in operation on several U.S. Navy Meteorology and Oceanography centers worldwide. Clients are deployed on great many sites throughout the U.S. Navy as well as U.S. Air Force, DoD, NATO, NOAA and other government agencies.

One particular source of original data is World Meteorological Organization's (WMO) data feed, containing a great number of land and sea surface and depth/height profile reports, forecasts, advisories, discussions, etc. – for the whole globe. A set of decoders processes the feed, and stores raw and decoded data in a database. A Metcast server distributes this information in an XML OMF format [3].

The Metcast server, the set of decoders for various WMO data formats, auxiliary and monitoring CGI scripts are all written in Scheme. Metcast clients are written in C++, Java, Scheme, Perl, Python, JavaScript, and Visual Basic.

The server and related modules are implemented in 12800 lines of Scheme code, counting the comments. WMO data feed decoders add 8400 more lines. The size of common extension libraries is 5400 lines of Scheme and some embedded C code. A Gambit-C 3.0 Scheme interpreter enhanced with compiled-in extensions has been used throughout the project.

2 Parsing of the data feed

Scheme proved to be particularly helpful in parsing of the WMO data feed. WMO code is a rather old, ad hoc, peculiar, somewhat inconsistent, tangled data format with a number of options, exceptions and special cases. Furthermore, received bulletins often contain errors due to manual miscoding and transmission problems.

A typical WMO report – for example, a surface synoptic report – is a sequence of code groups separated by white space. A code group is a string of letters, numbers and a few special characters. A code group or groups encode the result of observation of a particular quantity, e.g., cloud conditions, temperature, etc. If code groups were atomic tokens, a report could easily be parsed by a LR(1) automaton. Alas, code groups are composite entities that encode information in idiosyncratic ways. The mere identification of a code group depends on its position and context, which may encompass all previously seen code groups.

We have implemented a report decoder as a combination of a table-driven automaton and code-based group parsers. The latter recognize, parse, and validate a particular code group. The decoder takes a list of code groups and returns an associative list, an Abstract Syntax "Tree" (AST). A special procedure later walks the AST and records the parsed data in a database upload buffer. Of a particular help was Scheme's ability to store and pass procedural values as any other values. This let us implement decoders as *compositions* of code group parsers. For example, a very typical production `<a>?  <b>* <c>?` can be parsed by a combination `(sequence parse-a (sequence (loop parse-b) parse-c))`. This composition of group parsers is represented by a list `(parse-a (repetition-flag parse-b) parse-c)`. Given this list and the list of code groups to decode, a main driver walks both lists, applying the current parser to the current code

*This work has been supported by SPAWAR PMW-185, FNMOC and in part by the National Research Council, Naval Postgraduate School, and the Army Research Office under contracts 38690-MA and 40473-MA-SP.

group. The result of the application as well as the repetition flag determine if the current code group is consumed, if the next parser should be chosen, and how AST should be extended.

All the group parsers have the same interface. They receive as arguments the current code group and the AST, and should return:

- an association (a name-value pair) or a list of such associations to add to the AST;
- a symbol `pass` if the parser failed to recognize the code group. The code group should be given to the next parser;
- `#f` meaning a syntax error is detected at the current token;
- a symbol `terminate` to stop parsing of the report.

In the successful case (the first one above), the current token is assumed consumed. Any group parser may examine the AST (that is, the results of the previous parsers) and may even modify the AST. Therefore our parsing technique is somewhat similar to attribute grammars. Figure 1 shows an example of a group parser.

The example demonstrates an `and-let*` construction (SRFI-2), which was used frequently throughout the project and proved very helpful. As Fig. 1 shows, once the current token has been recognized as a potential <temperature-dew-point> group, `and-let*` carries on a sequence of elementary parsing decisions, all of which must succeed.

The Metcast decoder is continually processing incoming files, which are delivered every 1-3 minutes. A rather large batch of reports – 8 plain-text bulletins, 144 sea surface observation reports, 777 upper-air level data, 2 terminal airdrome forecasts and 322 synoptic reports – takes 8 wall-clock seconds to parse and 19 seconds to upload and record into the database. The platform is Sun Enterprise-450 server with two UltraSPARC-II CPUs and 512 MB RAM, running Solaris 2.6 and Informix 7.3 database. Keeping in mind that incoming reports have up to 10-minute delay from the time of issue, the total processing time at the Metcast end – under 1 minute – is entirely acceptable.

3 Implementing the Web application server

Scheme turned out to be a good implementation language for a web application server as well. One part of the server is a complex finite state machine that decides when a multipart reply is called for, and sends the corresponding MIME headers. The problem is not trivial as it is generally impossible to predict the number of non-empty replies for a complex request. Expressing such finite automata as sets of mutually-recursive procedures made the code clear and flexible.

Scheme was conducive to compilation and interpretation of the S-expression-based Metcast Request Language [2]. A request language phrase is compiled into a dictionary – an ordered sequence of bindings, – which constitutes the *environment* to look up all data needed to construct a Metcast database query. This hierarchical repository follows neither the static scope of Scheme expressions, nor the dynamic scope of procedure activations. Some bindings may be to procedures, which may push additional associations into the environment and thus affect further lookups.

Metcast server has a highly modular structure. The main program is responsible for receiving and parsing of a request, and packing of replies. Execution of a particular product request is delegated to a separate module (plug-in). The hierarchical repository was indispensable in implementing a *parameter bus,* which maintains the configuration for the main server and all plug-ins. The parameter bus also provides a uniform interface for invocation of modules and passing of a complex set of explicit and default parameters. For example, the main Metcast server module contains a form `(include "metar.scm")` that loads a plug-in metar.scm. The latter file defines procedures `perform-metar-request` and `perform-MSL-request`. The file binds these procedures to the corresponding Request Language verbs and the configuration information:

```
(env#bind*
 `((METAR (executor . ,perform-metar-request)
      (mime-type . "text/x-omf"))
   (MSL (executor . ,perform-MSL-request)
      (mime-type . "text/x-msl"))
   (OBJ-LOADER:st_constraint .
       ,(lambda constr-l
     (env#bind st_constraint constr-l))) ))
```

When metar.scm is loaded, the above initialization expression is evaluated. The Metcast server thus gains an ability to process requests for `METAR` and `MSL` products. The main server module contains a long chain of `(include "xxx.scm")` expressions, which define a set of requests a server accepts. Adding or replacing support for a particular product requests is as simple as loading or reloading the corresponding plug-in. This re-configuration and linking-in of the modules is possible while the server is running – although we have not pursued this opportunity. The flexible module linking mechanism was beneficial even in the static case as it made incremental development and evolution of the server easier.

4 Extensions to Scheme

Implementing Metcast required several extensions of the Gambit-C Scheme system: libraries of common procedures, and interfaces to external applications and the OS. Detailed descriptions for all extensions along with the commented source and validation code are freely available from a web site [4].

We have already mentioned one helpful extension: `and-let*`, an AND with local bindings, a guarded LET* special form. An input parsing library was another extension. It is a set of procedures that either skip, or build and return tokens following inclusion or delimiting semantics. The input parsing library has been used on very many occasions: in splitting WMO data feed files into bulletins and bulletins into code groups; in parsing of a `QUERY_STRING` or HTML form `POST` submissions; in breaking the response stream from a database query into rows and columns of data; in parsing of XML.

Another kind of extension – made possible by Gambit's excellent Foreign Function Interface – deals with accessing processes, files, directories, communication pipes and other objects external to a Scheme system. Scanning of a POSIX directory is implemented in a truly Scheme style and spirit: The OS:for-each-file-in-directory iterator combines the best features of `for-each`, `map`, and `filter`, and permits premature termination of iterations.

A very helpful extension that goes far beyond Scheme is opening and communicating through uni-, bi-directional, and TCP pipes as if they were regular files. This extension allows Scheme code to talk to external applications or

```
; <temperature-dew-point> ::= <temp> "/" <dew-point>?
; <temp> ::= "M"? <two-digits>    <dew-point> ::= "M"? <two-digits>
(lambda (token AST) ; "/" must be either in the pos 2 or 3
   (let ((slash-pos (string-index token #\/)))
     (if (not (memv slash-pos '(2 3))) 'pass
       (and-let*
         ((negate (lambda (x) (and x (- x))))
          (tempr
             (if (char=? #\M (string-ref token 0))
               (negate (string->integer token 1 3))
               (string->integer token 0 2)))
           (dp-pos (++ slash-pos))
           (dp (if (>= dp-pos (string-length token)) 'none
                  (if (char=? #\M (string-ref token dp-pos))
                    (negate (string->integer token (++ dp-pos) (+ 3 dp-pos)))
                    (string->integer token dp-pos (+ 2 dp-pos))))))
         (if (eq? dp 'none)
           (cons 'T tempr)
           (list (cons 'T tempr) (cons 'DP dp)))))))
```

Figure 1: A <temperature-dew-point> group parser

internet services. One particular kind of such an external application is a command-line SQL tool, which allowed us to build a portable database access library [4]. A database query interface is implemented in a Scheme spirit as well, as a general iterator over a collection of selected rows.

5 Illusory and real difficulties

Choosing an implementation language other than C or C++ inevitably raises the question of performance. We have run several benchmarks to ascertain the total performance and its contributing factors. For example, a sample request that retrieves 707 WMO messages (totaling 821K of output) took 25.6 sec (real), 24.1 sec (user) and under 0.1 sec of the system time. This running time comprises: loading and interpretation of the Metcast server script, database connection and query, Request Language interpretation, and output formatting. We conducted several experiments to isolate each of these factors, on the Sun E450 platform described above.

Connecting to a database with a SQL command-line tool dbaccess and running the query took 1.3 sec (real) and 1.0 sec (user). Thus the database interface – however ugly and inefficient it looks – is not the bottleneck. Parsing of the database reply in (interpreted) Scheme code adds 3.8 sec (real) and 2.2 sec (user) time. That is noticeable yet insignificant compared to the total time above. Instrumentation of the Metcast server showed that the server start-up time is under 1.0 sec of real time. This fact was one of the two biggest surprises. Given the complexity of the start-up process – launching of the Gambit interpreter, reading of the main script and 15 included scripts totaling 12800 lines of code, macro-expansion and byte-compilation – one would have expected the start-up to be a significant factor if not the bottleneck. The other biggest surprise was the fact that the most of the running time – 20 seconds – was spent within 7 lines of code, which copy characters from one stream to another while unescaping newlines. A makeshift optimization – copying streams line-by-line rather than character-by-character, and utilizing Gambit's undocumented function `##write-substring` – reduced the benchmark real running time from 25.6 sec down to 17.0 sec.

6 Conclusions

Implementation of a web application server and WMO decoders in Scheme showed that the language is up to the task. The elegance of Scheme and its ability to easily express guarded execution, finite-state machines as sets of mutually recursive actions, hierarchical repositories with procedural bindings turned out to be most important. Built-in garbage collection, iterators, safety, the ease of incremental testing cannot be overestimated either. Despite obvious inefficiencies, so far overall Metcast server performance is deemed satisfactory by customers.

References

[1] Oleg Kiselyov, "Distributing Weather Products through an HTTP pipe" `http://zowie.metnet.navy.mil/~spawar/JMV-TNG/` `http://pobox.com/~oleg/JMV-TNG/` March 10, 2000.

[2] Oleg Kiselyov, "A delegation language to request weather products and a scheme of its interpretation," Proc. third ACM SIGPLAN Int. Conf. on Functional Programming (ICFP'98), Baltimore, Maryland, Sep. 27-29, 1998, p. 343.

[3] Oleg Kiselyov, "Weather Observation Definition Format" `http://zowie.metnet.navy.mil/~spawar/JMV-TNG/XML/OMF.html` March 8, 2000.

[4] Oleg Kiselyov, "Scheme Hash," An archive of Scheme code `http://pobox.com/~oleg/ftp/Scheme/` July 4, 2000.