Implementing Metcast in Scheme

A Case Study

Metcast
http://zowie.metnet.navy.mil/~spawar/JMV-TNG/

- Request-reply and subscription system

- Database of real-time observation reports, forecasts, advisories, satellite imagery, gridded data from weather models, for the whole globe

- Domain-specific Request Language

- WMO data decoders (8400 lines)
  web application server, XML encoders and decoders, CGI scripts (12800 lines)
  common extension libraries (5400 lines)
  – are all in Scheme

This paper presents a case study of implementing a large distributed system in Scheme.

"Metcast" is a request-reply and subscription system for distributing, publishing and broadcasting of real-time weather information. The system stores a large amount of weather observation reports, forecasts, advisories, gridded data produced by weather models, as well as 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 (WMO) data feed, the Metcast server, XML encoders and decoders, auxiliary and monitoring CGI scripts are all written in Scheme. The numbers show the lines of code including the comments.

These are a couple of transparencies that I stole from my boss; he prepared them for a brief back in 1998, to people who might pay us money.

The Metcast system comprises clients and servers communicating in an HTTP protocol. 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. Some Metcast clients use the data to show pretty pictures, which would hopefully help a forecaster or a pilot. Other Metcast clients request data for use in various models: mission-planning, storm-tracking systems, etc. 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.

Next slide.

Here are a few examples of the pretty pictures. You can superimpose satellite imagery, isobar plots. Each dot shows a station. The dots are colored: a green dot means VFR flying conditions, a red one means IFR (that is, expect delays if you fly there).

Raw WMO feed

```
####018000176####
SACN31 KWBC 080000 RRB
METAR
CYFB 080000Z 13005KT 30SM BKN058 09/04
A2959 RMK SC7 SLP022=
CYMX 080000Z 23007KT 45SM FEW140 FEW250
15/11 A3004 RMK AC2CI0
SLP175=
...

####018000176####
SMCN04 CWAO 080000
AAXX 08004
71625 36964 00000 10176 20125 39996 40154
55007 333 10247 20050
70005=
71627 36964 32108 10200 20130 30126 40169
56016 333 10229 20104
70005=
...
```

One particular source of original data is a WMO data feed, containing a great number of land and sea surface and depth/height profile reports, forecasts, advisories, discussions, etc. This slide shows an excerpt from one of the feed file: weather conditions reported at Montreal on Sep. 7 (when I was preparing for this presentation). The first few reports at least look like written in English. The second group is not comprehensible at all without training.

OMF format


```
<!DOCTYPE OMF SYSTEM "http://zowie.metnet.navy.mil/~spawar/JMV-TNG/XML/OMF.dtd">
<Reports TStamp="968374366">
<SYN Title='METAR' TStamp='968371200'
LatLon='45.683, -74.033' BId='716278'
SName='CYMX, MONTREAL/MIRABEL' Elev='82'>
<SYID>CYMX 080000Z</SYID>
<SYCODE>RMK AC2CI0 SLP175</SYCODE>
<SYG T='15.0' TD='11.0' AS='1017.3' Vis='72450'
Ceiling='INF' Wind='230, 3.5' Clouds='22///'>
23007KT 45SM FEW140 FEW250 15/11 A3004
</SYG></SYN>
<SYN Title='AAXX' TStamp='968371200'
LatLon='45.467, -73.750' BId='716270'
SName='CYUL, MONTREAL/DORVAL' Elev='36' SType='AUTO'>
<SYID>71627</SYID>
<SYCODE>333 10229 20104 70005</SYCODE>
<SYG T='20.0' TD='13.0' P='1012.6' P0='1016.9'
Pd='6 -1.6' Vis='14000' Ceiling='INF'
Wind='210, 4' Prec=' ' Clouds='3////'>
36964 32108 10200 20130 30126 40169 56016
</SYG></SYN>
</Reports>
```

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. As you see, the data are indeed parsed, and the information is presented in a more comprehensible way: you can really tell the <u>temperature</u> and the <u>wind speed</u>. OMF as any XML document is aimed at an application rather than a human. One particular application, a CGI script written in Scheme, queries the Metcast server and turns the above XML document into the following

OMF to HTML translation

http://zowie.metnet.navy.mil/cgi-bin/oleg/get-wx-light

CYMX, MONTREAL/MIRABEL
Sep 08 00:00Z 45.683N 74.033W

Temp:	59 F
Alt:	30.04 Hg
Ceiling:	INFft
Cloud cover:	total: 2 lowest: 2 octas
Vis:	45 SM
Wind:	SW @ 7. KT

CYMX 080000Z
23007KT 45SM FEW140 FEW250 15/11 A3004
RMK AC2CI0 SLP175

simple neat HTML table. Note that temperature is converted to F. It is also clearer where the wind is blowing from. The HTML table is intended to be displayed on a PalmPilot VII, which has a wireless modem. This all works, btw, from the following URL. I won't talk this morning about neat things, only about ugly things.

WMO data parsing

<METAR-report> ::=
<report-type>? <station-id>
<report-modifier>? <timestamp>? <report-modifier>?
<wind-groups>? <horizontal-visibility-groups>?
<runway-visual-range>{0,4} <present-WX>{0,3}
<sky-condition-tokens>? <temperature-dew-point>?
<altimeter>? <remarks-section>?

<horizontal-visibility-groups> ::=
"CAVOK" | "9999" | "M1/4SM" | "<1/4SM" |
<fractional-SM> | <whole-SM> |
<whole-fractional-SM> | <whole-m> | <whole-km>

<fractional-SM> ::= "1/16SM" | "1/8SM" | ... "7/8SM"

<whole-SM> :: = <one-or-two-digits> "SM"

<whole-fractional-SM> ::=
<one-or-two-digits> " " <fractional-SM>
<fractional-SM> is a separate token

<whole-m> ::= <number> <compass-dir>?
(<number> <compass-dir>?)?

<whole-km> ::= <number> "KM"

Speaking of ugly things, here's a grammar for one of the 11 formats I'm parsing: land surface observations from fixed stations, in METAR code. Scheme proved to be particularly helpful in parsing of the WMO data feed. WMO code is a rather old, ad hoc, somewhat inconsistent data format with a number of options, exceptions and special cases. Furthermore, raw bulletins often contain errors due to manual miscoding and transmission problems.

Incidentally, this is a re-engineered grammar. The WMO manuals rely on a somewhat informal specification. The only purpose of this slide is to show what a tangle WMO data formats are.

As we see, a METAR 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, for example, horizontal visibility. Note, that many of the tokens can be omitted. Some of them may repeat more times than the manual on codes specifies, because it is the "common practice".

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 messy ways. The mere identification of a code group depends on its position and context, which may include all previously seen code groups.

Implementation of WMO decoders

Table-driven automaton + code-based group parsers

<a>? <b>* <c>? ⇒
((#f . parse-a) (#t . parse-b) (#f . parse-c))

A group parser:

- takes the current group and the AST
- returns:
  - (name . value) or a list of such associations to add to the AST
  - a symbol `pass`
  - `#f` on error
  - a symbol `terminate`

We have implemented a report decoder as a combination of a table-driven automaton and code-based group parsers. Group parsers recognize, decode, 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, this typical production is parsed by the following table: a list of pairs. The first element is the repetition flag, the second is a group parser.

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 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 those 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;

* etc.

In the successful case 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. Here's an example of a group parser.

Example of a group parser

```
(cons #f  ; <fractional-SM>
  (lambda (token AST)
    (let ((slash-pos
          (and (string-suffix? "SM" token)
            (string-index token #\/))))
      (if (not slash-pos) 'pass
        (LAND*
          ((num (string->integer token 0 slash-pos))
           (denom (string->integer token (++ slash-pos)
              (- (string-length token) 2)))
           ((< 0 num denom 17))
           (prev-vis-assoc (or (assq 'vis AST) '()))
           )
         (if (pair? prev-vis-assoc)
           (and (integer? (cdr prev-vis-assoc))
             (<= 1 (cdr prev-vis-assoc) 10)
             (begin
               (replace-ass! AST prev-vis-assoc
                (cons 'vis
                 (exact->inexact (* 1610
                   (+ (cdr prev-vis-assoc)
                      (/ num denom))))))
               '()))
           (cons 'vis (exact->inexact
              (/ (* 1610 num) denom)))) )))))
```

This parser handles the fractional SM production in the visibility group. #f means the production does not repeat. For the current 'token' to correspond to the fractional-SM production, the token should end in "SM" and contain a slash. If this is not the case, the current token is to be passed to the next group parser. If the current token does look like the fractional-SM non-terminal, we commit to parsing it. We determine the numerator and the denominator, and make sure they are reasonable (for example, none is zero). string->integer will return #f it it fails to parse a substring as an integer. In that case, LAND*, and-let*, skips the rest of the computation and immediately returns #f. This is the indication that the current token is in error. The main driver will print out this and the rest of the tokens, stop parsing of the report and record what has been parsed in the database, along with the indication of the parsing error and its place. If we managed to get the reasonable numerator and the denominator, we check the AST to see if we have encountered a visibility token. If we have, and it was an integer 1 through 10, that previous token is to be considered an integral part of the visibility group expressed in whole and fractional SM. We convert the entire number to meters and replace the already seen visibility. If the previous visibility token is an integer greater than 10, it tells then visibility in meters – and the current token is out of place and in error. If no visibility token has occurred before, we return an association to add to the AST. The main driver will do that, mark the current token as handled, and pass the next token to the next parser. I hope I made it obvious what a mess this all is. This is only one group parser; the METAR decoder contains 11 of them. The whole suite includes decoders for 11 various data formats.

Conclusions

Particular Benefits of Scheme re. Metcast

- Ease of blending data- and code-driven execution (e.g., context-sensitive parsing)
- Guarded execution
- Ease of expressing FSMs
- Hierarchical repositories with procedural bindings
- Parameter bus – *dynamic typelinking*
- Satisfactory performance

Let me state the particular benefits of Scheme that become apparent while implementing Metcast. These benefits are *in addition* to safety, the ease of incremental testing, powerful iterators and macro system, and garbage collection. The particular benefits are: the ease of blending data- and code-driven execution, which greatly helped in implementing context-sensitive parsing in a flexible and clear way. SRFI-2, a guarded LET* form, was used on many occasions throughout the project and proved very helpful. 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 multi-part 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.

The article gives a few details about the performance. I have talked about the first few points. Now that I have said conclusions, I can talk about the Metcast Module system without a fear of being cut short.

Module system of Metcast server

server.scm:

```
(include "metar.scm")
(include "sat-serve.scm")
(include "grid-serve.scm")
...
```

metar.scm:

```
(define (perform-metar-request . prod-parms)
   ; parse parameters in the local (METAR) scope
   ; and load them into the current env
  (if (pair? prod-parms)
    (for-each parse-request-line prod-parms))
  (let*
    ((bounding-box (env#demand bounding-box))
     (modified-since (env#find modified-since))
     (mime-type (or (env#find mime-type)
                    "text/plain")) ...))
```

Module system (cont'd)

```
(define (perform-MSL-request . prod-parms)
...)


(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)))

))
```

Metcast server has a highly modular structure. The main server is responsible for receiving and parsing of a request, and packing of replies. Execution of a particular product request is delegated to separate 'modules' (plug-ins). The configuration for the main server and all plug-ins is maintained in a hierarchical repository, a 'parameter bus'. 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 server 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 configuration information (default value for the mime-type parameter).

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 initialization expression also defines a new datatype, st_constraint, and tells the server how to load it. 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 module. 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 static configuration as it made the incremental development and evolution of the server easier.

Extensions to Scheme

- and-let* (SRFI-2)
- An input parsing library
- POSIX interfaces
- OS:for-each-file-in-directory
- Uni-, bi-directional, and TCP pipes as files

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. We've already seen an example of using and-let*, or LAND*, which was used on many occasions. Another very helpful library extension was an input parsing library: It is a set of procedures that either skip, or build and return tokens following inclusion or delimiting semantics. The library was used for parsing of bulletins, QUERY_STRING, XML, etc. 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 internet services.