Impressions of the ICFP'08 Programming Contest

Peter Salvi, Huang Jianshi
Agenda

• What is ICFP?
• Overview of the contest
• This year's problem
• Solution outline
• LISP in action
• Demo
• Summary
ICFP

• International Conference on Functional Programming
• Annual programming contest (since 1998)
  − Results made public at the conference
• Declarations of “honor”:
  − 1st place: The programming language of choice for discriminating hackers
  − 2nd place: A fine tool for many applications
  − Lightning division: Very suitable for rapid prototyping
Previous Contests – 2004

- Organizer: University of Pennsylvania;
  Ant colony with state-machine ants
Previous Contests – 2005

• Organizer: PLT Group;
  Cop & Robber bot programming
Previous Contests – 2006

• Organizer: Carnegie Mellon University; Decipher and emulate the ancient codex machine (UMIX), then solve the problems left by the ancient people.
Previous Contests – 2007

- Organizer: Utrecht University; Help an alien to acclimatize by altering its DNA-string with a two-stage virtual machine
This year's contest

• July 11 – 14 (Friday (Saturday) – Monday)
• Organizer: Portland State University & University of Chicago
• Theme: Guide a Martian rover on hostile terrain to its home base through a TCP/IP connection
• 24 hours for the lightning round
• Submit binaries for a Linux LiveCD

... and thus the team Epsilon was formed...
Organization

• Wiki pages (e.g. FAQ)
• Mailing list
• IRC channel
• RSS feed of the changes on the homepage
• Graphical server for the rover (written in SML)
• While the contest was running:
  - Task description made more clear
  - New programs for the LiveCD
  - Bugfixes for the server
Programming Languages

• Results announced at ICFP'08 (Sept. 22-24)
• Several videos and slides on the net
• 336 submission (+ 140 lightning round)
• Languages:
  • Java, Python, C++
  • Haskell, ML-family
  • ...
  • Lisp (only 7)
  • Many others (LaTeX (!))
Participants

• Participants from various countries
• Japan: 106 (!) [USA: 192]
The Problem

- Communicate with the rover by TCP/IP
- Information rate: about 10 messages / second
- Messages contain terrain data:
  - Boulders, craters and Martians (everything circular)
  - Elliptical view
Vehicle Model

- Control: turn left / turn right / accelerate / brake
- The rover is a double state machine:
Map

- On every map, there are five runs, with different starting positions
- Only the best three counts
- Home base is at the center
- Map size, number of objects and other parameters vary
Theory of a Solution

• Modules:
  - Communication
  - Parser
  - Mapper
  - Route Planner
  - Vehicle Controller
  - Logger / Visualizer (for debugging)

• Go from abstract to concrete
Route Planner

• Simplest method: just go for the home base
• We actually used this, with modifications:
  - If there is some obstruction ahead, go for the closest of the two tangent points on the perimeter

Martians are treated as circular objects (the radius is a parameter depending on its visible speed)
Route Planner

• Problematic case:

• Solution: When both directions are blocked, it tries to turn left/right until there is no obstruction in a given distance

• Remembers the direction it has chosen

• Good points:
  - No “drunk driver” effect
  - Simple & fast, straightforward method
Route Planner

• Real problems:
  − Martians are simplified too much
    • Approximate by ellipses (just a bit more complex)
    • Do a real simulation and evasion (time-consuming)
  − Only one point is considered (no real planning)
  − The specified destinations may not be reachable
Route Planner

• We can use an A* search
  − The nodes are points of a dynamic grid
  − Guarantees that we can reach the base (if the world is known)
Route Planner

• We can use an A* search
  - The nodes are points of a dynamic grid
  - Guarantees that we can reach the base (if the world is known)
Mapper

- **Stores**
  - All persistent objects (= not Martians)
  - Martians
    - Only the last few
    - In a fixed-size queue
    - Recent Martians are remembered even if not visible

- **Storage method**
  - Simple list (for simple planning)
  - Dynamic grid (for A* search)
Dynamic Grid

- Quadtree (2D binary tree)
- Fixed number of points in every cell
- For every object there should be points at a short distance
Motion Control

• Actual movement is calculated by
  - Speed \(S_t\)
  - Acceleration / Breaking \(a\), init. value unknown
  - Drag coefficient \(k\), init. value unknown

\[
s_{t'} = \max(s_t + (t' - t)a - k(t' - t)s_t^2, 0)
\]

• The angle can be computed by:
  - Soft turn speed
  - Hard turn speed
Motion Control

- Goal:
  - Make the rover move along the path as fast as possible within acceptable errors.

![Diagram of motion control system](image-url)
Motion Control

• The 3 main parts of the controller:
  - Rover's movement model
  - Input / Output
  - Control algorithm
Motion Control

• Model
  - The rover model is ideal (as specified in the task)
  - The motion equation:

\[
\begin{align*}
    s' &= v \\
    v' &= a \\
    \phi' &= \omega \\
    \omega' &= \alpha
\end{align*}
\]

\[
s_t' = \max(s_t + (t' - t)a - k(t' - t)s_t^2, 0)
\]
Motion Control

- **Input**
  - Distance to the path
  - Angle to the path's tangent line

- **Output**
  - Acceleration
  - Angular acceleration
Motion Control

• We didn't solve any DEs... : )

• Simulation-based control algorithm
  - Simple and effective
  - Proportional gain is enough
  - Less parameter tuning
  - But more computation-expensive (not a problem)
Motion Control

• Other

  ▪ Finally the controller converts the numerical values to commands that the rover can understand

  ▪ Parameter tuning:
    • Only trial-and-error
    • Most important parameters:
      ▪ Simulation period
      ▪ Threshold for the soft / hard turn
Motion Control

• Problems
  − May oscillate at sudden turns
  − We do not brake
    • We want to go fast!
    • The solution space would become two-dimensional (an optimization algorithm is preferred than hand-tuning)
Messages

• Every message consists of:
  − An identifier (one character)
  − Data (objects are divided by yet another identifier)
  − Semicolon

• Objects are messages without a semicolon

• I $dx \ dy \ time-limit \ min-sensor \ max-sensor \ ... \ ;$
• T $time-samp \ vehicle-ctl \ ... \ object^* \ ;$
• b $x \ y \ radius$
• m $x \ y \ direction \ speed$
Internal Message Format

• A message like

\text{T 123 aL ... b 13.5 23.47 4.3 m 3.2 4 45 4.1 ;}

... would be rendered as

\begin{verbatim}
(telemetry
 (time . 123)
 (control-state . (accelerate hard-left)))
...
(objects
 (boulder (x . 13.5d0) (y . 23.47d0) (radius . 4.3d0))
 (martian (x . 3.2d0) (y . 4.0d0)
 (direction . 45.0d0) (speed . 4.1d0))))
\end{verbatim}
We want to program like this:

(defpaser telemetry \T t (stream)
  (time (read stream))
  (control-state (read-control-state stream))
  ...
  (objects (iter (for next = (peek-char t stream)))
    (until (char= next #\;))
    (collect (parse-stream stream)))))

(defpaser boulder \b nil (stream)
  (x (read-float stream))
  (y (read-float stream))
  (radius (read-float stream)))

(defpaser martian \m nil (stream)
  (x (read-float stream))
  (y (read-float stream))
  (direction (read-float stream))
  (speed (read-float stream)))
Parser

- We would like an expansion like this:

```lisp
(progn
 ((defun parser-telemetry (stream)
   (prog1 (let* ((time (read stream))
                 (control-state (read-control-state stream))
                 ...
                 (objects (iter (for next = (peek-char t stream))
                               (until (char= next "\";))
                               (collect (parse-stream stream)))))))
   (list (cons 'time time)
         (cons 'control-state control-state)
         ...
         (cons 'objects objects)))
   (check-semicolon stream)))
 (setf (gethash "\t" *parser-table*)
       (cons 'telemetry #'parser-telemetry))
)
```

The result is an alist of the data

Hash table of the message handlers

Takes a semicolon or gives an error
The macro:

```lisp
(defmacro defparser (name type semicolon-terminated (stream)
   &body name-value-pairs)
  (let ((fname (concatenated-symbol 'parser- name)))
    (progn
      (defun ,fname (,stream)
        ,(if semicolon-terminated
            (progl (create-alist ,name-value-pairs)
                (check-semicolon ,stream)
                (create-alist ,name-value-pairs)))
          (setf (gethash ,type *parser-table*) (cons ',name #',fname))))))

(defmacro create-alist (pairs)
  `(let* ,pairs
     (list @(iter (for var in (mapcar #'first pairs))
            (collect `(cons ',var ,var)))))))```
The main parser is very easy now:

```
(defun parse-stream (stream)
  "Parses STREAM using the parsers in *PARSER-TABLE*."  
  (unless (peek-char t stream nil nil) (throw 'exit 'done))
  (let* ((type (read-char stream))
         (parser (gethash type *parser-table*))
         (if parser
             (cons (car parser) (funcall (cdr parser) stream))
             (error "No parser for message type ~c." type)))))
```

... of course, this is just one step; higher levels of abstractions can be built over this
Logging

• Very important for debugging
• Should be able to
  - Turn off instantly (with no efficiency drawback)
  - Select logging method
  - Visualize (later)
• Perfect chance to use macros
  - Even in C(++) it is usually done by macros:
    #ifdef DEBUG
    ...
    #endif
(defparameter *logging* t)

(defun rover-controller-main-loop ()
    (with-logs ((mapping :filename "/tmp/rover-map.log"
                        :options (rover martians))
             (control :stream *error-output*))
        ...) )

(defun mapper (...)
    ...
    (write-log (s mapping (rover))
        (format s "Rover position: "a"%" " ...))
    (write-log (s mapping (martians))
        (format s "Martian position: "a"%" " ...))
    (write-log (s mapping (rover martians))
        (format s "Rover-Martian distance: "a"%" " ...))
    ...
)
Logging Macro - Properties

• Change (and recompile) only some main function to refine the logging parameters
  − Where does the log go
  − What subsets should be logged

• Set *LOGGING* to NIL and recompile everything, and there will be no trace of logging left

• WITH-LOGS just calls WITH-LOG recursively:

```lisp
(defun with-logs (logging-descriptions &body body)
  (if (null logging-descriptions)
      `(progn ,@body)
      `(with-log ,'(first logging-descriptions)
                (with-logs ,'(rest logging-descriptions) ,@body))))
```
Logging Macro

- The setup macro:

```lisp
(defmacro with-log ((name &key stream filename options) &body body)
  (when *logging*
    `(unwind-protect
       (progn
         ,(if (null filename)
             (cons `(setf (gethash ',name *log-hash*)
                     (cons ,stream ',options))
                   body)
             (let ((s (gensym)))
               `(with-open-file ,s ,filename
                 ((with-open-file ,s ,filename
                   :direction :output
                   :if-exists :supersede
                   :if-does-not-exist :create)
                     (setf (gethash ',name *log-hash*)
                           (cons ,s ',options))
                           ,@body))))))))
  (remhash ',name *log-hash*)))))
```
Logging Macro

- And the logging macro:

```
(defmacro write-log ((stream name &optional dependencies) &body body)
  (when *logging*
    (let ((log-with-options (gensym)))
      `(let (((log-with-options (gethash ',name *log-hash*))
        (when (and ,log-with-options
          (every (lambda (option)
            (member option (cdr ,log-with-options))))
          ',dependencies))
        (let ((,stream (car ,log-with-options))
          ,@body)))))))
```

- Simple, but very efficient
- Less code duplication, more control
PostScript Logs

• “Graphical logs” are easy with PostScript
• PostScript is a stack language, like Forth
PostScript Logs

• Now define some colors and set the map size:

% Colors:
/boulder { 0.4 0.2 0 } def
/visibleBoulder { 0.7 0.5 0 } def
/crater { 0.3 0.3 0.3 } def
/visibleCrater { 0.6 0.6 0.6 } def
/martian { 1 0 0 } def
/rover { 0 0 1 } def
/home { homeRadius 0 1 0 } def

% Coordinate system transformation
/mapSize 300 def
/setupMap { 297.5 421 translate 595 mapSize div dup scale } def

• This allows us to write simple definitions for the objects on the map
PostScript Logs

• The actual logs look like this:

```
%%Page: t=1 1
setupMap
0 0 home circle
30 30 25 25 rover vehicle
-2 14 2 13 martian vehicle
20 23 4 visibleBoulder circle
29 -4 5 crater circle
showpage
```

```
%%Page: t=2 2
setupMap
0 0 home circle
25 22 25 20 rover vehicle
2 13 6 14 martian vehicle
20 23 4 visibleBoulder circle
16 7 5 visibleCrater circle
29 -4 5 crater circle
showpage
```
PostScript Logs

• The output:
Log Visualization

• ... but on the contest we have used CL-SDL
• The logs were output in a format that can be read (almost) directly as a list of CLOS objects
• The $k$th line of the log is of the format:

\[
(frame \ k \\
  \begin{array}{c}
  (rover \ :x \ _ \ :y \ _ \ :dest-x \ _ \ :dest-y \ _) \\
  (martian \ :x \ _ \ :y \ _ \ :dest-x \ _ \ :dest-y \ _) \\
  (boulder \ :x \ _ \ :y \ _ \ :radius \ _ \ :visible-p \ _) \\
  \ldots
\end{array}
\]

... where ROVER, MARTIAN, BOULDER, etc. are all CLOS class names
Log Visualization

- Read with READ and call MAKE-INSTANCE on its children to create the objects
- In the main loop, just read a frame and call a display method on every object
- Optimization: log only new objects (ie. Objects not seen before and Martians)
- The whole visualization environment, including everything, is about 100 lines of code
Conclusion

• We have used SBCL to generate an executable
• Our rover only got to the 7th map
• But it was a lot of fun!
• Next year @ Edinburgh!
  - http://icfpconference.org/
  - Maybe with more Lispers?
• Slides (English and Japanese) can be found at:

  http://www.den.rcast.u-tokyo.ac.jp/~salvi/archives/text.html
Thank you for your attention!