Program Mobile Robots in Scheme[1]

Jonathan Rees   Bruce Donald
Computer Science Robotics and Vision Laboratory
Cornell University
Ithaca, NY 14850

Abstract

We have implemented a software environment that permits a small mobile robot to be programmed using the Scheme programming language[3]. The environment supports incremental modifications to running programs and interactive debugging using a distributed read-evaluate-print loop. To ensure that the programming environment consumes a minimum of the robot's scarce on-board resources, it separates the essential on-board run-time system from the development environment, which runs on a separate workstation. The development environment takes advantage of the workstation's large address space and user environment. It is fully detachable, so that the robot can operate autonomously if desired, and can be reattached for retrospective analysis of the robot's behavior.

To make concurrent applications easier to write, the run-time library provides multitasking and synchronization primitives. Tasks are light-weight and all tasks run in the same address space. Although the programming environment was designed with one particular mobile robot architecture in mind, it is in principle applicable to other embedded systems.

1 Introduction

The Lisp family of programming languages has a long history as a basis for rapid prototyping of complex systems and experimentation in new programming paradigms. Polymorphism, higher-order procedures, automatic memory management, and the ability to use a functional programming style all aid the development of concise and reliable programs[1]. In this paper we promote the application of the Scheme dialect of Lisp to programming a mobile robot system whose limited resources might contraindicate the use of high-level language features and an integrated programming environment.

The following example gives the flavor of the Scheme system. The procedure `sonar-accumulate` takes as its arguments an initial state and a procedure that combines a previous state with a sonar reading to obtain a new state. The combination procedure is called once for each of the twelve sonar transducers, and is supplied with the transducer number and the sensed distance. The returned value is the final state.

```
(define (sonar-accumulate combine init)
  (let loop ((val init)
             (i 0))
    (if (>= i 6)
        val
        (let* ((j (+ i 6))
               (rs (read-sonars i j)))
          (loop (combine j (cdr rs)
                         (combine i (car rs)
                                  val))
                (+ i 1))))))
```

The implementation of this control abstraction hides the fact that the sonar hardware allows only particular transducer pairs to be read simultaneously. `sonar-accumulate` might be used to determine which transducer is giving the smallest reading:

```
(define (nearest-sonar)
  (sonar-accumulate
        (lambda (i dist previous)
          (if (< dist (cdr previous))
              (cons i dist)
              previous))
        (cons -1 infinity)))
```

[1] This paper describes research done in the Computer Science Robotics and Vision Laboratory at Cornell University. Support for our robotics research is provided in part by the National Science Foundation under grants no. IRI-8802390 and IRI-9000532 and by a Presidential Young Investigator award to Bruce Donald, and in part by the Air Force Office of Sponsored Research, the Mathematical Sciences Institute, Intel Corporation, and AT&T Bell Laboratories.

Published in *Proceedings of the 1992 IEEE International Conference on Robotics and Automation,* pages 2681-2688. This reprint incorporates some minor corrections.

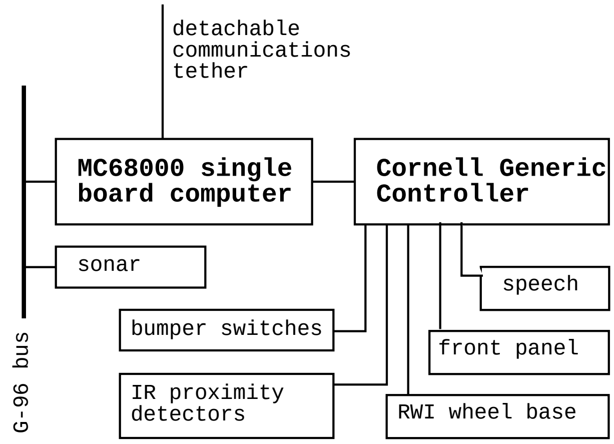

Figure 2: Hardware architecture.

Figure 1: Tommy, the mobile robot.

The paper is structured as follows: Sections 2 and 3 describe the hardware and software architecture of the robot and its Scheme implementation. Section 4 briefly presents facilities available to the programmer beyond what Scheme ordinarily provides. Section 5 discusses the relative success of the design and ways in which it could have been different. Section 6 describes what we would like to do next. An appendix gives an extended example.

2 Hardware

The particular hardware targeted by this project is a Cornell mobile robot built on the Real World Interface B12 wheel base (figures 1 and 2). The robot is roughly cylindrical, about 50 cm tall with a 30 cm diameter. An enclosure contains a rack in which is mounted several processor and I/O boards. The robot architecture is distributed and modular, so that different sensors and effectors are easily added and removed. Interprocessor communications is via 19.2 Kbaud serial lines. Low-level I/O is handled by a Cornell Generic Controller (CGC), a general-purpose control board based on an Intel 80C196 processor[10]. High-level task control and planning are performed by Scheme programs running on an off-the-shelf Motorola 68000 processor board (Gespak MPL 4080). The 68000 board has .5 Mbyte RAM and .25 Mbyte EPROM, and Scheme currently uses no off-board memory.

The entire robot is powered by rechargeable batteries in the wheel base. The robot draws about 1.2 amps when idle, more when moving.

Much of Scheme's communication with sensors and effectors is accomplished by messages transmitted to and received from a Cornell Generic Controller over a serial line. The CGC relays messages to and from various other devices. In principle Scheme could communicate directly with many devices as it does with the sonar, but we prefer to off-load device control to the CGC, which has richer I/O capabilities.

The features of this architecture relevant to Scheme are:

- Small physical size (one 10 by 17 by 1 cm circuit board) — this means small memory size compared to a workstation.
- Low power consumption — this means a slow processor (16 MHz 68000 board, 130 mA).
- Low bandwidth for communications with the workstation.

Also part of the hardware for the overall development system is a workstation capable of running a full-sized Scheme or Common Lisp implementation, text editor, and so on; currently this is a Sun Sparcstation, but any similar workstation would work as well. The workstation communicates with Scheme on the robot over a 19.2 Kbaud serial line tether. Some kind of wireless communication would be nice, but we are

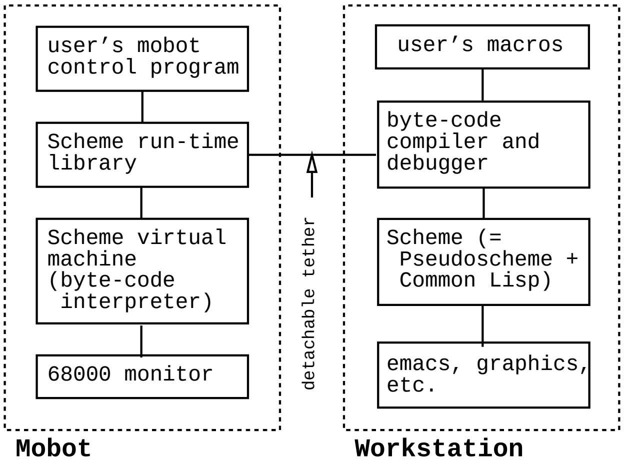

Figure 3: Software architecture.

concerned about robustness and dropout, and would like to maintain the option of running autonomously.

3 Scheme system architecture

The Scheme system consists of a run-time environment, which resides on the robot, and a development environment, which resides on a separate workstation. The two communicate with each other over a serial line. Figure 3 summarizes the major software components of the Scheme system.

3.1 Run-time environment

The run-time environment builds on the Scheme48 virtual machine architecture[12]. The virtual machine is byte-code based and stack-oriented, closely resembling the target machine of the Scheme 311 compiler[4], and similar in spirit to [8]. The VM handles memory management: allocation instructions (such as `cons`) can cause garbage collections. The virtual machine is implemented by an interpreter and garbage collector written in C. Compiled for the 68000, the virtual machine consumes about 24 Kbytes of EPROM.

The VM has I/O instructions corresponding to the Scheme procedures `read-char` and `write-char`. These instructions are executed using traps to a simple supervisor-mode monitor. Access to the appropriate sonar control registers also requires a small amount of virtual machine support. Other than Scheme and the monitor, no other operating system runs on the 68000.

The virtual machine executes both user code and software for communication with the development system. The communications software and a standard run-time library (see section 4) reside in EPROM, while byte codes for user programs are downloaded from the workstation over the tether.

3.2 Development environment

The programmer using the Scheme system interacts with the development environment, which runs under a Scheme implementation on a workstation. For Scheme on the workstation, we are currently running Pseudoscheme[14] under Lucid Common Lisp, but MIT Scheme or any of various other Scheme implementations would work as well. We could run Scheme48 itself on the workstation, but Lucid and MIT Scheme are preferable for their speed (they both sport optimizing native-code compilers) and for their integration with other software on the workstation, including existing packages for graphics, planning, computational geometry, and spatial reasoning.

The development environment is an 11 Mbyte executable image, and usually runs under the control of GNU Emacs. It includes a command loop that accepts Scheme expressions to evaluate and other commands that control the environment in various ways: load a source file, reset the run-time system, etc. The development system translates Scheme source code into a byte-code instruction stream, which is transmitted over the serial line to the Scheme run-time system on the 68000.

The development system performs as much preprocessing as possible on user code before sending the code to the run-time system. The compilation process includes symbol table lookups, so no variable names or tables need to reside in the robot itself. Error messages and backtraces that come back from the run-time system are interpreted relative to the same symbol tables resident in the development system. As a result of this policy, on-board overhead is kept to a minimum. The run-time library, which includes communications software, standard Scheme library procedures such as `append` and `string->number`, and the extensions described below, is only about 70 Kbytes of byte codes and data.

When the tether is connected, it is possible for a user program on the robot to call procedures on the workstation, and vice versa. For example, a program running on the robot can initiate graphics routines that display output on the workstation's monitor.

4 Run-time library

Besides standard Scheme procedures such as `+` and `vector-ref`, our Scheme comes equipped with procedures that support sensor/effector control, multitasking, and remote procedure call.

4.1 Controlling sensors and effectors

The run-time library contains a set of procedures for obtaining information from sensors and for issuing commands to effectors. For example, `(read-sonar 7)` reads sonar unit number 7, and `(translate-relative 250)` instructs the wheel base to initiate a 250 mm forward motion. At a low level, the various devices use different units and coordinate systems, but the library converts to consistent units for use by Scheme programs.

Most sensor and effector control is mediated by a CGC. The run-time library contains routines written in Scheme that communicate with the CGC over a serial line. Most operations consist of a single message exchange. Access to the serial lines and sonar hardware is synchronized to prevent conflicts when several threads perform different operations concurrently.

4.2 Lightweight threads

Multitasking is useful in writing programs that simultaneously manage several different input and output devices, or sensors and effectors in the case of a robot. Our Scheme environment supports light-weight threads with a library routine `spawn`. The argument to `spawn` is a procedure of no arguments. The call to `spawn` returns immediately, and a new thread is started concurrently to execute a call to the procedure. All threads run in the same address space, so threads may communicate with one another by writing and reading shared variables or data structures.

For example, it may be convenient to have a dead-reckoning integrator running continuously in its own thread:

```
(define (reckon-loop)
  (let ((o1 *current-odometry*)
        (o2 (read-odometers)))
    (set! *current-configuration*
          (integrate-configuration
             *current-configuration*
             o1
             o2))
    (set! *current-odometry* o2))
  (sleep *reckon-interval*)
  (reckon-loop))

(spawn reckon-loop)
```

Other threads may consult the integrator's estimation of the current configuration (x, y, θ) by simply consulting the value of `*current-configuration*`. Because variable references and assignments (and in fact, all virtual machine instructions) are atomic, any particular reference to this variable will yield a consistent configuration triple.

The implementation of threads is written entirely in Scheme, that is, above the level of the virtual machine. This is made possible, even easy, by the existence in Scheme of the `call-with-current-continuation` primitive, which the virtual machine implements efficiently. See [16] for an elegant presentation of this technique for building an operating system kernel. (Actually, thread switching uses the same low-level continuation mechanism that `call-with-current-continuation` does, but does not interact with `dynamic-wind`, described in the appendix.)

Because threads are ordinary Scheme objects with no special status in the virtual machine, they are garbage collected when no longer accessible. (All currently runnable threads are of course accessible.)

Synchronization is provided by lock operations similar to those in Zetalisp or Lucid Common Lisp: `(make-lock)` creates a new lock, and `(with-lock` *lock thunk*`)` calls *thunk* after obtaining control of the lock, blocking if some other thread currently holds the lock. The lock is released on any normal or exceptional return from the call to *thunk*. A second form of synchronization is condition variables as in [15], which are single-assignment storage locations. A condition variable is created with `make-condvar`, assigned with `condvar-ref`, and set (at most once) with `condvar-set!`. A `condvar-ref` that precedes a condition variable's `condvar-set!` blocks until some other thread performs such a `condvar-set!`.

4.3 Remote procedure call

Our Scheme system also supports a simple remote procedure call capability. Whenever the tether is attached, a procedure on the mobot may call a procedure on the workstation, or vice versa. Because address spaces are not shared, the mechanism is not transparent, but it is supported by the procedures `host-procedure` and `mobot-procedure`. E.g.

```
(define plan-path
  (host-procedure 'plan-path))

... (plan-path destination) ...
```

This defines a procedure `plan-path` on the robot that, when called, initiates an RPC to the procedure of the same name residing on the host. This might be desirable if, say, `plan-path` were too slow or too large to run on the robot itself. Displaying graphical output is another use for RPC's from the robot to the workstation.

Similarly, a Scheme program on the workstation may call procedures on the robot using `mobot-procedure`. For example, robot sensor and effector routines are easily accessed from the workstation:

```
(define read-all-infrareds
  (mobot-procedure 'read-all-infrareds))

... (read-all-infrareds) ...
```

Multiple threads on host and robot may perform remote procedure calls concurrently.

5 Discussion

Our thesis is that the combination of a high level language with rapid turnaround for changes allows for more experiments with the robot per unit time. As with many claims about software engineering, this is difficult to test in any objective way. But we think this will turn out to be true, as it has apparently been true when Lisp and Scheme have been applied to other domains.

There is nothing new about programming robots in Lisp. To our knowledge, however, our implementation is unique in providing an advanced on-board Lisp environment for a small robot.

Why implement Scheme? There exist good cross-compilers and cross-debuggers for C, and these are in many ways well suited for developing embedded systems. However:

- We prefer Scheme to C because its high-level features (polymorphism, automatic memory management, and higher-order procedures) allow for more concise, reliable programs.

- Because of its neutral syntax and powerful macro and code-manipulation facilities, Scheme and Lisp have historically been a good base for experimentation with special-purpose languages.

- The conventional architecture for a C programming environment requires compiling entire modules and linking the entire program every time a change is made to the program. With compiled programs being sent over a 19.2 Kbaud serial line, this makes the turnaround time for changes unacceptably slow; and the alternative of putting the C compiler and linker on the robot itself would make the robot too large.

Another possible choice for a general-purpose robot programming language would have been ML (or Concurrent ML[15]). Scheme made more sense for us given the educational background of the students and researchers that use the robot. Also, the Scheme48 system already existed when this project started, and was already well suited for cross-development; adapting an existing ML implementation would have been much more work for us.

What about real-time constraints? Languages with garbage collection have traditionally suffered from delays of up to several seconds while memory is being reclaimed. We take the approach that one can live with short delays. Essential tasks that require that there be no garbage collection delays can run on separate processors that do not run garbage collectors (e.g. they can be programmed in C). However, the fact that the 68000's memory is nonvirtual and fixed size means that we can put an upper bound on the amount of time taken by a garbage collection, and limiting the amount of live data will limit the frequency of garbage collection; thus we can get absolute time bounds for specific tasks, even when they run as Scheme programs that allocate memory. With our current garbage collector, a garbage collection of a 50% full heap requires over half a second. This is slow, but we believe that this time can be improved upon by tuning the code or by switching to a generational collector[13].

Our choice of synchronization primitives is merely conventional, not necessarily final. The set is not really sufficient in that it does not include an easy way to wait on multiple events. We experimented with Concurrent ML's primitives[15], but found that programs using them were difficult to debug. The `future` construct[9] would be easy to implement in the Scheme virtual machine, but is probably inappropriate in this context, since its purpose is exploiting parallelism, not programming embedded systems.

While there has been some work on special-purpose robot control languages (see e.g. [11] and review in [7]), we considered it safer to hedge our bets by putting our effort into building an uncommitted general-purpose infrastructure.

6 Future work

Members of the Cornell Robotics and Vision Laboratory are using this Scheme system for prototyping a variety of navigation, planning, and manipulation systems. In particular, we intend to use the mobile robot for testing a mathematical theory of task-directed sensing and planning[6]. As the Scheme system continues to be exercised, opportunities to improve the programming infrastructure will continue to arise. The communications software needs to be made more robust, and further debugging aids, including a trace package and inspector, need to be implemented.

The imminent arrival of additional robots running Scheme will raise interesting issues in developing control programs for collaboration. It will be possible to use a single host environment for coordinated debugging of multiple robot systems.

We would like to experiment with and compare various programming language constructs and paradigms for describing robot control systems. Scheme should be an ideal medium for this. Of particular interest to us are subsumption architecture[2], ALFA[7], and Amala[5].

Performance may be a problem in the future. Any interpreter for a virtual instruction set is likely to be 20 to 30 times slower than equivalent code compiled for the target hardware. If the interpreter turns out to be a bottleneck, we'll consider using a Scheme compiler, either an existing one (Scheme-to-C, MIT Scheme, etc.) or a new one. The advantages of doing so must be weighed against the effect of lower density of native code relative to the virtual instruction set. This might be an important consideration given current memory limitations.

Acknowledgments

Thanks to Loretta Pompilio for drawing the illustration in figure 1.

Thanks to Craig Becker, Russell Brown, and Jim Jennings for making the robot hardware and software work, and for fruitful discussions about the architecture of the Scheme implementation.

Thanks to Norman Adams and Richard Kelsey for their comments on a draft of this paper.

Richard Kelsey is coauthor with Jonathan Rees of Scheme48. We appreciate his help in getting Scheme48 to run on the robot.

The virtual machine was cross-compiled and cross-linked using software developed and supported by the Free Software Foundation. The availability of source code was a great help in making our cross-development environment work.

References

[1] Harold Abelson and Gerald Jay Sussman. Lisp: A language for stratified design. *BYTE,* February 1988, pages 207–218.

[2] Rodney A. Brooks. A robust layered control system for a mobile robot. *IEEE JRA* RA-2 (1):14–23, 1986.

[3] William Clinger and Jonathan A. Rees, editors. The revised^4 report on the algorithmic language Scheme. *Lisp Pointers* 4(3), ACM, 1991.

[4] William Clinger. The Scheme 311 compiler: An exercise in denotational semantics. In *Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming,* pages 356–364.

[5] Mike Dixon. *Embedded Computation and the Semantics of Programs.* PhD thesis, Stanford University, 1991.

[6] Bruce Donald and Jim Jennings. Constructive recognizability for task-directed robot programming. In *Proceedings of the 1992 IEEE International Conference on Robotics and Automation.*

[7] Erann Gat. ALFA: A language for programming reactive robotics control systems. *Proceedings of the 1991 IEEE International Conference on Robotics and Automation,* pages 1116–1121.

[8] Adele Goldberg and David Robson. *Smalltalk-80: The Language and its Implementation.* Addison-Wesley, 1983.

[9] Robert H. Halstead. Multilisp: a language for concurrent symbolic computation. *ACM Transactions on Programming Languages and Systems* 7(4):501–538, October 1985.

[10] Jim Jennings. Modular software and hardware for robot construction. Unpublished manuscript.

[11] Leslie Pack Kaelbling and Stanley J. Rosenstein. Action and planning in embedded agents. In *Designing Autonomous Agents: Theory and Practice from Biology to Engineering and Back,* MIT Press, 1990.

[12] Richard Kelsey and Jonathan Rees. Scheme48 progress report. Manuscript in preparation.

[13] Henry Lieberman and Carl Hewitt. A real-time garbage collector based on the lifetimes of objects. *Communications of the ACM* 26(6): 419–429, 1983.

[14] Jonathan Rees. A Scheme to Common Lisp translator. Manuscript in preparation.

[15] John H. Reppy. CML: a higher-order concurrent language. In *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation,* pages 293–305, 1991.

[16] Mitchell Wand. Continuation-based multiprocessing. In *Conference Record of the 1980 Lisp Conference,* pages 19–28. The Lisp Conference, P.O. Box 487, Redwood Estates CA, 1980. Proceedings reprinted by ACM.

Appendix

This appendix gives an extended example of robot programming in Scheme, illustrating the use of threads and higher-order procedures. `heel` is a simple loop that causes the robot to follow whatever is in front of it. Most of the rest of the example consists of a guarded servo loop that modulates the pulse width commanded to the translation and rotation motors.

```
(define (heel)
  (kill-motors-on-exit
   (lambda ()
     (let loop ()
       (let ((l (read-sonar 0))
             (r (read-sonar 11)))
         (translate (- (min l r)
                       *target-distance*))
         (rotate (if (< r l) -15 15))
         (sleep *heel-delay*)
         (loop))))))
```

```
(define *heel-delay* (round/ one-second 4))
(define *target-distance* 400)
```

`kill-motors-on-exit` uses `dynamic-wind` to ensure that any exceptional exit will turn off the motors. `dynamic-wind` is a run-time procedure also found in some other Scheme dialects and analogous to Common Lisp's `unwind-protect`. Its three arguments are an *entry thunk,* a *body thunk,* and *exit thunk,* all of which are procedures of no arguments. Ordinarily the three are invoked in order, and the value delivered by the body thunk is returned. The exit thunk is also called on exceptional exits (more precisely, invocations of externally created continuations) from the call to the body thunk, and the entry thunk is called on exceptional re-entry, an unusual situation made possible by `call-with-current-continuation`.

```
(define (kill-motors-on-exit thunk)
  (dynamic-wind (lambda () #f)
                thunk
                (lambda ()
                  (translate-stop)
                  (rotate-stop))))
```

`make-servo-loop` is a higher-order procedure that can be passed a particular odometer and motor and control constants, returning a procedure that will control that motor. A control procedure produced by `make-servo-loop` is called with a target velocity and a guard, and returns when the guard predicate returns true. The guard is a procedure that is passed the current odometer reading, an estimated velocity, and the most recent power level that was commanded to the motor. `friction` is the amount of power that has to be applied in order to overcome static friction, `drag` is the amount of power required to increase velocity by 1 mm/sec, and `hysteresis` = 1/ (time required for a power increase of 1 unit to have its full effect on velocity).

```
(define (make-servo-loop odometer apply-power
                         friction drag
                         hysteresis)
  (lambda (velocity guard)
    (dynamic-wind
      (lambda () #f)
      (lambda ()
        (let loop ((x1 (odometer))
                   (t1 (time))
                   (pw (if (> velocity 0)
                           friction
                           (- 0 friction)))
                   (update-at (time)))
          (apply-power pw)
          (let ((t2 (time))
                (x2 (odometer)))
            (let ((v
                   (round/ (* (- x2 x1) one-second)
                           (- t2 t1))))
              (or (guard x1 v pw)
                  (if (< t2 update-at)
                      (loop x2 t2 pw update-at)
                      (let ((dpower
                             (* (- velocity v)
                                drag)))
                        (loop x2 t2
                              (+ pw dpower)
                              (+ t2
                                 (round/
                                    (abs dpower)
                                    hysteresis))))
                      ))))))
      (lambda () (apply-power 0)))))
```

concurrent-servo-loop is a combinator that takes a synchronous servo loop and returns a procedure that will spawn a server loop to run asynchronously. The logic here is tricky because if there's already a servo loop running, it must be terminated before the new one can start.

```
(define (concurrent-servo-loop synchronous-loop)
  (let ((lock (make-lock))
        (condvar (make-condvar))
        (stop? #f))
    (condvar-set! condvar 'idle)
    (lambda (velocity guard)
      (with-lock lock
        (lambda ()
          (set! stop? 'usurp)
          (condvar-ref condvar)
          (set! condvar (make-condvar))
          (spawn
           (lambda ()
             (dynamic-wind
              (lambda () (set! stop? #f))
              (lambda ()
                (synchronous-loop
                    velocity
                    (disjoin (lambda (x v power)
                               stop?)
                             guard)))
              (lambda ()
                (condvar-set! condvar
                              'done)))))))))))

; Constants determined by experiment.
(define translate-until
  (concurrent-servo-loop
    (make-servo-loop translate-where
                     apply-translate-power
                     6199  ;static friction
                     14    ;drag
                     12))) ;hysteresis

(define (translate dist)
  (let ((target (+ (translate-where) dist)))
    (if (>= dist 0)
        (translate-until
           *usual-speed*
           (disjoin (position-guard > target)
                    bumper-guard
                    impediment-guard))
        (translate-until
           (- 0 *usual-speed*)
           (disjoin (position-guard < target)
                    impediment-guard)))))

(define *usual-speed* 120)

(define (translate-stop)
  (translate-until 0 (lambda ignore 'stop)))
```

```
(define rotate-until
  (concurrent-servo-loop
    (make-servo-loop rotate-where
                     apply-rotate-power
                     4449  ;static friction
                     160   ;drag
                     16))) ;hysteresis
```

rotate and rotate-stop are similar to translate and translate-stop.

```
; Guard against going past a given position.
(define (position-guard pred value)
  (lambda (x v power)
    (if (pred x value)
        'arrived
        #f)))

; Guard against trying to go beyond an impediment.
(define (impediment-guard x v power)
  (if (and (= x 0)
           (> (abs power) *impediment-power*))
      'impediment
      #f))

(define *impediment-power* 15000) ;ca. 200 watts

; Guard against hitting things with the bumper.
(define (bumper-guard x v power)
  (if (> (read-bumpers) 0)
      'bumper
      #f))

; Combine a set of guards into a single guard.
(define (disjoin guard . guards)
  (if (null? guards)
      guard
      (let ((g (apply disjoin guards)))
        (lambda args
          (or (apply guard args)
              (apply g args))))))

(define (apply-translate-power power)
  (translate-power (power->pulse-width power)))

(define (apply-rotate-power power)
  (rotate-power (power->pulse-width power)))

(define (power->pulse-width power)
  (let ((pw (round/ power (battery-voltage))))
    (if (< (abs pw) *pulse-width-limit*)
        pw
        (error "commanded power is too high"
               power))))

(define *pulse-width-limit* 128)
```