GRL Primer

GRL or "GiRL" is a high level language for writing robot control policies. GRL programs are structured like electronic circuits (or neural networks) in that they consist of networks of signals, which are the equivalents of variables. Like variables in standard programming languages, signals have types and values. GRL is strongly typed, meaning that the type of a signal can't change over time. Its value, on the other hand, is recomputed continually, so it typically changes over time. The GRL compiler performs type propagation, so you typically don't need to declare the types of your signals; The compiler figures them out.

GRL is embedded in Scheme. That means that it runs inside of Scheme; You can type GRL code at the Scheme prompt, change things you don't like, and when you're ready, have the GRL compiler compile it for you. The compiler expands all your signals, computes their types, does a lot of optimization, and spits it all as either Scheme code or some other language that you can compile for the robot of your choice.

A quick example

You make signals with the define-signal form. At the Scheme prompt (or in a .scm file that you load), just type:

```
(define-signal signal-name signal-expression)
```

Where signal-name is the name of the signal and signal-expression is a Scheme-like expression for computing the value of the signal from the values of other signals. For example, if we have a set of signals called freespace-left, freespace-right, and freespace-ahead, then we could write a control policy for a freespace follower as:

```
;; Parameters
(define-signal stopping-distance 10)
(define-signal rotate-gain 1.5)
(define-signal translate-gain 2.1)

;; Actual control code
(define-signal angle-error (- freespace-left freespace-right))
(define-signal clearance-error (- freespace-ahead stopping-distance))

(define-signal rotate-command (* rotate-gain angle-error))
(define-signal translate-command (* translate-gain clearance-error))
```

1 “Form” is just a fancy term for command or expression.
Here are some comments and clarifications, in case you're confused about it:

- The `;;` notation is the way you mark a comment in Scheme. Everything from the first `;;` in a line to the end of the line is a comment.

  1. I've put the names of the signals being defined in **boldface** just to make it easier to read.

- You may wonder why constants like `rotate-gain` are defined as "signals" even though they never change. That's just so that we have nice symbolic names for constants rather than having strange numbers floating around the code with no explanation. It doesn't matter a lot in a short program like this, but when you write a large program, it will matter a lot.

- **Advanced students:** what `(define-signal a b)` really does is create a Scheme data structure to represent the signal `b`, and then binds it to the Scheme variable `a`. If you type:

```
.inspect a
```

at the Scheme prompt, it will show you all the slots in the compile-time representation of the signal, such as its `inputs` (a list of signals), its `operator` (procedure), its `type`, its `declared type`, etc. This can be useful for tracking down compile-time errors.

This code defines a simulated parallel circuit that looks like this:

![Parallel Circuit Diagram]

If you feed this program above to the compiler by typing:

```
(compile-to-c rotate-command translate-command)
```

Then the compiler will generate something like the following:

```c
void run() {
    int freespace_left;
    int freespace_right;
    int freespace_ahead;
    float rotate_command;
    float translate_command;

    while (1) {
        update_girl_time();
        before_signal_update();
        ... code to compute the values of freespace_left, freespace_right, and freespace_ahead ...
        rotate_command = 1.5*(freespace_left-freespace_right);
        translate_command = 2.1*(freespace_ahead-10);
        after_signal_update();
    }
}
```
It generates an infinite loop that first calls a couple of internal functions in the GRL runtime system (`update_girl_time()` and `before_signal_update()`), then computes the new values of the signals, and finally calls another internal function, `after_signal_update()`. Don't worry about the fact that it didn't bother to generate variables for most of the signals. That's because the compiler only cares about generating the values of the signals you specify in the compile command, in this case, `rotate-command` and `translate-command`. It noticed that the other signals were only used once so it just included the code for computing them in the code for computing `rotate-command` and `translate-command`.

Now suppose we add the definition:

```lisp
(define-signal base-controller
  (drive-base rotate-command translate-command))
```

Which gives us a circuit that looks like:

Here we assume that `drive-base` is a function that that's already been defined to call a pair of C procedures, `set_rotate_velocity` and `set_translate_velocity`, which changes the low-level settings of the robot hardware. If we then said:

```lisp
(compile-to-c base-controller)
```

we would get the following:

```c
void run() {
  int freespace_left;
  int freespace_right;
  int freespace_ahead;

  while (1) {
    update_girl_time();
    before_signal_update();
    ... code to compute the values of freespace_left, freespace_right, and freespace_ahead ...
    set_rotate_velocity(1.5*(freespace_left-freespace_right));
    set_translate_velocity(2.1*(freespace_ahead-10));
    after_signal_update();
  }
}
```

Some more comments and clarifications in case you're confused:

- Drive-base doesn't really return a value (it's type is `void`) and so base-controller is only a signal in a degenerate sense. That's why the compiler doesn't bother to generate a variable named base-controller.
- The only purpose of the base-controller signal is to have something to pass to compile.
Again, don't worry about the fact that it hasn't generated variables for the rotate and translate commands. They weren't needed. All that matters is that the right values were passed to set_rotate_velocity and set_translate_velocity.

If you ran this program on the robot, it would try to drive forward and it would veer left and right, depending on what direction had the most free-space. Unfortunately, there's a very unfortunate case where the robot drives up to a wall. Then the robot stops driving forward because freespace_ahead is small, but it also stops turning because freespace_left and freespace_right are virtually identical. So the robot just sits there. Since freespace_left and freespace_right probably aren't exactly the same, the robot will turn very slowly one way or another until it finally has turned parallel to the wall and can drive forward. However, that can take a very long time and it makes your robot look really stupid.

Because of this problem, people often make the rotate gain bigger when the robot is "stuck" like that. We can code that in GRL by just changing the definition of rotate-gain:

```
(define-signal stuck-distance 15)
(define-signal stuck? (< freespace-ahead stuck-distance))
(define-signal rotate-gain
  (if stuck?
   999999
   1.5))
```

This gives you a slightly more complicated circuit:
```c
void run() {
    int freespace_left;
    int freespace_right;
    int freespace_ahead;
    float rotate_gain;

    while (1) {
        update_girl_time();
        before_signal_update();
        ... code to compute the values of freespace_left, freespace_right, and freespace_ahead ...
        if (freespace_ahead<15) {
            rotate_gain = 999999;
        } else {
            rotate_gain = 1.5;
        }
        set_rotate_velocity(rotate_gain*
                           (freespace_left-freespace_right));
        set_translate_velocity(2.1*(freespace_ahead-10));
        after_signal_update();
    }
}
```

The problem with this program is that if someone walks in front of the robot, there’s likely to be a brief period when they’re directly in front of the robot and so the \texttt{freespace\_ahead} signal will be very small. During that time, the rotate speed would effectively be infinite and the robot would suddenly turn in place, possibly right into a wall. This is not a good thing. A common solution to this problem is to put a timer into the code so that the robot has to be stuck for some minimum period of time before (say, 800ms), before it raises the gain. In C or Scheme, this wouldn’t be too bad to write, but it would be a non-trivial amount of work: you’d have to add a new local variable to hold the timer, you’d have to make sure the timer got updated at the right times and reset at the right times, etc. In GRL, you can just use the \texttt{true-time} function, which returns the number of milliseconds for which its input has been true:

```
(define-signal stuck?
    (> (true-time (< freespace\_ahead stuck\_distance))
        800))
```

which gives us the circuit:

Now when we compile the program, we get something a little more sophisticated:
```cpp
void run() {
    int freespace_left;
    int freespace_right;
    int freespace_ahead;
    float rotate_gain;
    int stuckP_onset=0;

    while (1) {
        update_girl_time();
        before_signal_update();
        // code to compute the values of freespace_left, freespace_right, and freespace_ahead ...
        if (! (freespace_ahead<15)) {
            stuckP_onset = ms_clock;
        } else {
            rotate_gain = 1.5;
        }
        set_rotate_velocity(rotate_gain*
            (freespace_left - freespace_right));
        set_translate_velocity(2.1*(freespace_ahead - 10));
        after_signal_update();
    }
}
```

This ability to write functions that have internal state turns out to be really useful in writing robot control programs. While it's possible to do equivalent things in C++ using objects or in Scheme using closures, the code ends up looking somewhat klunkier because you have to manually create separate copies of the timer as closures or objects:

```cpp
class timer {
public:
    int true_time(bool predicate) {
        if (predicate)
            onset = ms_clock;
        return ms_clock-onset;
    }
private:
    int onset;
}

void run() {
    int freespace_left;
    int freespace_right;
    int freespace_ahead;
    float rotate_gain;
    timer stuck_timer;

    while (1) {
        update_girl_time();
        before_signal_update();
        // code to compute the values of freespace_left, freespace_right, and freespace_ahead ...
        if (stuck_timer.true_time(freespace_ahead<15)>800) {
            rotate_gain = 999999;
        } else {
```

Copyright © 1999, 2000 Ian Douglas Horswill. All Rights Reserved.
rotate_gain = 1.5;
}
set_rotate_velocity(1.5*(freespace_left-freespace_right));
set_translate_velocity(2.1*(freespace_ahead-10));
}
}

Data structures

For various reasons that will become clear later in the course, it’s somewhat inconvenient to have `rwi-base-controller` take the rotation and translation velocities as separate arguments. We’d prefer to package them together as a single data structure. GRL supports record structures called groups. The easiest way to create a group is with `define-group-type`:

```scheme
(define-group-type name
   constructor
   (member-name accessor-name) ...)
```

For example:

```scheme
(define-group-type rt-vector
   (rt-vector rotation translation)
   (rotation rotation-of)
   (translation translation-of))
```

This says that `rt-vectors` are a new kind of group, that you make a new `rt-vector` by saying

```scheme
(rt-vector rotation-exp translation-exp)
```

where `rotation-exp` and `translation-exp` are the signals that you want to be the rotation and translation components of the group, and that you extract the rotation and translation signals from a group by using the `rotation-of` and `translation-of` functions. If we assume that `drive-base` has been modified to take an `rt-vector` as an argument rather than a pair of arguments, then we can just change the definition of `base-controller` to:

```scheme
(define-signal base-controller
   (drive-base
    (rt-vector rotate-command
                translate-command)))
```

Why would we want to do this? After all, it seems like we’ve increased the amount we had to type without actually increasing the clarity of the code (in fact, we’ve decreased it). One answer is that you often find yourself wanting to write several different control policies and then sum their outputs. Unfortunately, the control policies each really have two outputs: the rotation command and the translation command. One of the features of GRL groups is that the compiler automatically maps primitive procedures across the elements of the group so that if you say:

```scheme
(define-signal policy1 (rt-vector ... rotation code ... ... translation code ...)
(define-signal policy2 (rt-vector ... different rotation code ...
                       ... different translation code ...)
```

then you can just say:
(define-signal base-controller
  (drive-base (+ policy1 policy2)))

which would generate a rotation and translation which was the sum of the respective rotation and translations of the two policies:

![Diagram]

Which we will usually just notate as:

![Diagram]

We can also do more complicated functions of policies, such as:

(define-signal base-controller
  (drive-base (+ (* policy1 7.8)
                policy2)))

which would give us a weighted sum, i.e.:

![Diagram]

or even:

(define-signal base-controller
  (drive-base (+ (* policy1
                  (if stuck?
                      3.8
                      0.2))
              policy2)))

which would give us a weighted sum with variable weights:
or to be really strange:

\[(\text{define-signal}\ \text{base-controller}\n\quad (\text{drive-base}\ (\text{max}\ \text{policy1}\ \text{policy2})))\]

which would give us a rotation which was the greater of the two rotations of the two policies, and a translation which was the greater of the translations of the two policies. Since this is sort of a silly control policy, I won’t bother drawing it as a circuit.

**Signal procedures**

GRL lets you operate on signals with most of the standard Scheme arithmetic procedures (+, -, *, abs, max, sin, cos, etc.), as well as conditionals (if), vector operations (make-vector, vector, vector-ref), min, max, arg-min, arg-max, bitwise operations (bitwise-and, bitwise-ior, etc.), I/O (read, display), list operations (with serious restrictions), and a few GRL-specific primitives. Of course, it wouldn’t be a very useful language if it didn’t allow you to define new ones. GRL has a few different things that act like procedures. The most common are called *signal procedures*. Signal procedures are defined more or less the same way procedures are defined in Scheme: either with lambda, as in:

\[(\lambda (a\ b\ c)\n\quad (/* (+ a\ b\ c)\n\quad 3))\]

or by specifying a set of arguments in define-signal:

\[(\text{define-signal}\ (\text{mean-3}\ a\ b\ c)\n\quad ((/^\ (+ a\ b\ c)\n\quad 3))\]

you can then apply it to (i.e. call it on) signals as you like:

\[(- (\text{mean-3}\ a\ b\ c)\n\quad (\text{mean-3}\ d\ e\ f))\]

Signal procedures get called at *compile time*, that is, while the compiler is running. Their job is to take their input signals, which are chunks of parallel circuitry, as arguments, and return a new, usually bigger signal as a result. Therefore, he compiled circuit for the expression (mean3 a b c) looks like:
And the compiled circuit for \((- (\text{mean-3 } a \ b \ c) (\text{mean-3 } d \ e \ f))\) looks like:

However, we will usually draw it in its unexpanded form:

Since it is more compact and easier to understand:

\text{Mean-3} is a badly designed procedure, however, because it only takes three arguments. If we want to take the mean of different numbers of arguments, we have to write \text{mean-1}, \text{mean-2}, \text{mean-3}, \text{etc.} and then have to keep track of how many arguments we’re call it with. Fortunately, GRL supports procedures with variable numbers of arguments using the same "." notation used in Scheme:

\begin{verbatim}
(define-signal (mean . args)
  (/ (apply + args) (length args)))
\end{verbatim}

The first line here says that \text{mean} is a procedure that supports a variable number of arguments and that those arguments should be bound (as a list) to the variable \text{args}. The call to \text{apply} in the second line says to call the + procedure and pass it all the arguments in \text{args}. Thus \text{mean} is a signal procedure that
we can call with any number of signals as arguments and it will return a signal which is the mean of the argument signals.

Notes for advanced students:
- All the processing of variable numbers of arguments, and of calls to `apply` and `length` are handled by the compiler at compile time. Thus an expression like `(mean a b c)` will get compiled into the C code: `(a+b+c)/3`. The compiler never generates code that dynamically allocates memory, performs list operations, or requires run-time type checking. It is therefore safe to compile into C or BASIC. This also means that it is as fast or faster than hand-written C or BASIC code.
- You are therefore not allowed to have a signals whose run-time values are lists.
- Signal-procedures are allowed to be recursive as long as the recursion can be expanded at compile time into an inlined expression. What this means in practice is that the only things you can recurse on are lists. You can't write something like the recursive gcd algorithm or Newton's method because the compiler can't determine how many times to expand the recursion without knowing the run-time values of the signals.
- Like signals, signal procedures are just Scheme data structures that get manipulated by the compiler. However, they contain within them real Scheme procedures that do the actual work of taking the arguments to the signal procedure and constructing a new signal from them.

The GRL library

GRL includes a large library of useful operations that are sufficient for most people’s programming needs. Here is a partial list of the operations it supports:

- Time-series analysis: integrals and derivatives, one-shots, detection of onset and termination of signals, hysteresis
- Data storage: flip-flops, latches, toggles
- Signal conditioning: low-pass filtering, median filtering clipping and saturation, dead-zones
- Angle operations: addition, subtraction, normalization
- Vector operations: sum, mean, reduce, min, max, arg-min/max, dot products, prefix scans, shifting and rotating elements.
- Other: counters, signal-reduce, …

The library also supports a number of standard data types, including:

- Polar and Cartesian vector operations and conversion between them
- Representation of lines in 2-space and computation of intersection between them
- Intervals with union, intersection, etc.
- Sampled functions: given a function return a vector of values for the function at different argument values. If the function and sampling interval are compile-time constants, the vector will also be reduced to a constant at compile-time, so it is useful for automatic generation of lookup tables.

There are also a separate library for writing behavior-based systems (behavior-utilities), which includes data types for behaviors and standard combinators for behaviors, such as motor-schema combination and subsumption.

You should look over the documentation for both these libraries at some point during the class.

Advanced topics

Thus far, we’ve only talked about the most basic features of the language. There are a number of more sophisticated things you can do that you’ll probably want to learn at some point. Some of these will be used in the course notes.
Transducers

It's important to understand that when you say:

```
(define-signal foo
  (+ (* a 2)
    (- b 7)))
```

in GRL, the arguments to `+` are not numbers. The arguments are signals and the result is another signal. The result signal can then be passed the compiler, which will generate code that performs an addition at run time, and that addition will take numbers as its arguments. But at compile time, when you're typing GRL code, the arguments to `+` are really signals and the kinds of expressions you can put in `define-signal` forms are limited. What you're really saying is something more like:

```
(define foo (map-signals +
  (map-signals * a (make-constant-signal 2))
  (map-signals - b (make-constant-signal 7))))
```

This is the real Scheme code that's being run by the Scheme interpreter. The `(+ (* a 2) (- b 7)))` version is called a signal-expression and it's transformed into the Scheme expression by a macro called signal-expression. The result is a data structure of type signal, and that data structure gets walked by the compiler to reconstruct the C expression `foo=a*2+(b-7)`.

You can include most of the Scheme features you would want in signal expressions: `let`, `let*`, `letrec`, `cond`, `case`, `lambda`, `apply`, most numeric procedures, `car`, `cdr`, `length`, `list`, etc. You cannot, however, include things like `set!` or loops because they would be meaningless: the loops and `set!`s would run at compile time, not at run time.

However, there are times when you really want to be able to specify some raw Scheme code to run at compile time, though, and GRL does allow you to do this. While signal-procedures are defined in terms of signal-expressions, there is another kind of function-like object called a transducer that can be defined in terms of raw Scheme code. Here's an example of a transducer:

```
(define-transducer (sum-over-time input)
  (state-variables (sum 0))
  (set! sum (+ sum input))
  sum)
```

The first line says that `sum-over-time` is a transducer and that it has a local variable (called a state variable) called `sum`. The state variable will get initialized once at the start of the program to 0. After that, it will be updated once per cycle of the while loop using the update rule: `(set! sum (+ sum input))`. The return value of the transducer is the sum. If we call the transducer twice:

```
(define-signal a (+ 7 (sum-over-time b)))
(define-signal c (+ 8 (sum-over-time d)))
```

and compile it, we get something like:

```
void run() {
  int a;
  int b;
  int c;
  int d;
  int sum=0;
```
int sum_2=0;

while (1) {
    update_girl_time();
    before_signal_update();
    ... code to compute the values of b, and d...

    ;; Update the sum-over-time transducer for b
    sum = sum+b;
    ;; Update a
    a = 7+sum;

    ;; Update the sum-over-time transducer for d
    sum_2 = sum_2+d;
    ;; Update c
    c = 8+sum_2;
    after_signal_update();
}

Transducers are allowed to contain (almost) arbitrary Scheme code. In particular, you have loops, state
variables, and assignment statements. You can't have recursion, or list operations, however, nor can you
include GRL code like define-signal inside of a transducer.

We'll see examples of more complicated transducers later on in the class. However, much of the point of
the GRL language is that you don't have to write a lot of transducers. There are a few standard ones in a
library (sum, difference, integrate, differentiate, count, delay, one-shot, true-time, low-pass-filter, run-length, hysteresis) that can usually combined to do whatever you need. For now, it will suffice for you just to know what transducers are and to have some idea how to
read the code for one.

**Vectors**

GRL also supports vectors. A signal can have a vector as a value, so long as the compiler can determine its
length and element type at compile time. You can create a vector by calling the vector function, which
returns a vector whose elements are the various arguments to vector, in order. Thus:

```
(define-signal direction
    (vector x y))
```

would create a signal, direction, whose type was “(vector integer 2)”, i.e. a vector of two
integers, and would insure that its first element was always the value of the signal x, and its second was
always the value of the signal y. Another way to make a vector is with make-vector. The call

```
(make-vector length element-value)
```

returns a length-element vector, all of whose elements are element-value. If element-value is a time-varying signal, then all the elements of the vector will change along with that signal. However, length must be a constant; you can’t dynamically resize vectors at run-time.

You can operate on vectors using vector-ref and vector-length, as in Scheme. However, it is
more common to operate on whole vectors at once. As with groups, primitive operations are automatically
mapped across the elements of vectors. So if you add to vectors, you get a vector which is the sum of their
respective elements. If you add a constant to a vector, you get a vector whose elements of the elements of
the first vector plus the constant. For example, you could find out what elements of a vector were greater
than average by first finding the mean of the elements of a vector, and then comparing each element to the mean. In GRL, you could do this with the signal expression:

```
(> my-vector (vector-mean my-vector))
```

The call to `vector-mean` returns a (scalar) signal whose value at any point in time is the mean of the values of the elements of `my-vector` at that point in time. The call to `>` then compares each of the elements of `my-vector` to that value and returns a Boolean. The result is a vector of Booleans telling which values are above average.

Here's a more complicated example. Suppose you want to notice when someone is walking up to the robot by noticing when one of the sonars is returning a decreasing reading. Let's assume we have two vector signals, `sonar-distances`, and `old-sonar-distances`, which give the values of the sonars now and at the previous clock tick. By subtracting the elements of `sonar-distances` from `old-sonar-distances`, we can get a number that is large when a given element is decreasing and small when it isn't. We can then ask for the index of the element that's decreasing fastest by saying:

```
(vector-arg-max (- old-sonar-distances sonar-distances))
```

where `vector-arg-max` is a predefined transducer that reads the elements of a vector and returns the index of its largest element. There are a number of useful vector routines build into the GRL library, including procedures for shifting and rotating the elements of vectors, finding minima and maxima, and reducing a procedure over the elements of a vector. To find out more, check the documentation on the [GRL library](#).

Here's a good example of vector processing in GRL. Suppose we have a vector of readings from a ring of sonars and we want to convert them to a vector of Cartesian coordinates of the points in space from which their respective echoes were received. To start with, we need to know in what direction the different sonars are pointing. We'll assume that they're pointed in equally spaced directions in a ring. So if there are `k` sonars, then the orientation of sonar `i` is \( 2\pi i/k \) radians. The GRL primitive `(index-generator k)` returns a vector of elements numbered successively from 0 to `k-1`. So the signal expression

```
(define-signal sonar-orientations
  (/* 2 pi (index-generator sonar-count))
  sonar-count))
```

Will return a vector of orientations of each of the sonars, expressed in radians. A unit vector in a given direction is given by

```
(define-signal (unit-vector theta)
  (xy-vector (cos theta)
            (sin theta)))
```

or, to be somewhat more fancy:

```
(define-signal (unit-vector theta)
  ((xy-vector cos sin)
   theta))
```

(Remember that operations get mapped over groups – including function application, so applying a group of functions to a value returns the group of results). The definition:

```
(define-signal sonar-unit-vectors
  (unit-vector sonar-orientations))
```
which expands to:

```
(define-signal sonar-unit-vectors
  ((xy-vector cos sin)
   (/ (* 2 pi (index-generator sonar-count))
      sonar-count)))
```

then gives us a vector of unit vectors for the different sonars.\(^2\) Using the function sampling routines in the GRL library, would could also have written this more clearly as:

```
(define-signal sonar-unit-vectors
  (sample (xy-vector cos sin)
          (interval 0 (* 2 pi))
          sonar-count)
```

which says “sonar-unit-vectors is a vector of sonar-count xy-vectors of the cosine and sine of the angles from 0 to 2π.”

Having computed the unit vectors, we can now compute the Cartesian positions (in egocentric, or robot-body-centered, coordinates) of the objects generating the sonar echos by multiplying the unit vectors by the distances returned by the sonars:

```
(define-signal egocentric-ping-positions
  (* sonar-distances
     (sample (xy-vector cos sin)
             (interval 0 (* 2 pi))
             sonar-count))
```

This compiles into roughly the C code:

```
for (i=0; i<sonar_count; i++) {
  egocentric_ping_positions_x[i] = cos(2*pi*i/sonar_count) * sonar_readings[i];
  egocentric_ping_positions_y[i] = sin(2*pi*i/sonar_count) * sonar_readings[i];
}
```

except that the compiler will realize that all the calls to cos and sin can be computed off-line at compile time and place in a lookup-table:

```
float temp_sin[16] = { 0.0000, ... };
float temp_cos[16] = { 1.0000, ... };
for (i=0; i<16; i++) {
  egocentric_ping_positions_x[i] = temp_cos[i] * sonar_readings[i];
  egocentric_ping_positions_y[i] = temp_sin[i] * sonar_readings[i];
}
```

(Here we are assuming that sonar-count is 16).

---

\(^2\) Technically, it returns an xy-vector whose components are each normal vectors, rather than a vector of xy-vectors. However, the two representations can be treated interchangeably.
Recursion

Recursive signal procedures

There are two different types of recursion in GRL. There’s the normal computer science kind of recursion, where a signal procedure calls itself. Recursive signal procedures are allowed in GRL and are common, in fact. The only restriction on this kind of recursion is that since signal procedures are expanded at compile time, the signal procedure has to be written so that it will terminate at compile time. That means that the choice between recursing and terminating the recursion can’t be based on the value of a signal, which usually isn’t known until run-time. This means that you can’t write functions like factorial directly in GRL. If you tried to:

```
(define-signal (factorial n)
  (if (zero? n)
      1
      (* (factorial (- n 1))
         n)))
```

then the compiler would go into an infinite loop trying to expand the recursive call to `factorial`. Fortunately, no one has ever complained about being unable to use factorial in a GRL program. If, however, you really need to use something like factorial, you can always write it by hand in C (or whatever target language you are using) and then call it from a transducer.

The main time you use recursive signal procedures in GRL is for operating on lists. For example, the signal procedure:

```
(define-signal (signal-reduce operator signal-list)
  (if (= (length signal-list) 1)
      (car signal-list)
      (operator (car signal-list)
                (signal-reduce operator (cdr signal-list)))))
```

works fine because all the list operations (car, cdr, and length) get run at compile-time, and therefore the test in the if is a compile-time constant. This allows the compiler to expand only one arm of the if.

If you compile the signal-expression `(signal-reduce + (list a b c))`, the compiler first evaluates the call to list, then calls signal-reduce. Signal-reduce creates a signal with if as an operator and the signal `(= (length (list a b c)) 1)` as a test, then returns that signal as its result. The compiler then expands the arguments to the if signal. In the process of expanding the = signal, it encounters the call to length and evaluates it immediately, reducing it to 3. It then has the signal `(= 3 1)`, which it can reduce to #f, which then allows it to reduce the if signal to its second arm and throw the other away. In the process of expanding the second arm, it will encounter the recursive call to signal-reduce. This will again expand into a recursive call, but with a shorter list. When that recursive call is expanded by the compiler, it will find the call to = to be true. The compiler will then take the first arm of the if, which has no recursive call, which means it can terminate the expansion process. The result is the circuit:

```
+   +
|   |
|   |
a   b
```

Copyright © 1999, 2000 Ian Douglas Horswill. All Rights Reserved.
Of course, this is a bad example because if one really wanted to add the values of a list of signals, one could just use apply. We could equally well have used a complicated signal procedure instead of +. However, using + as the operator argument keeps the result simple.

**Recursive signals (a.k.a. recurrence relations, recurrent networks, cyclic circuits)**

The other kind of recursion in GRL is what electrical engineering calls recursion – a signal whose current value is defined in terms of its previous value and some other input(s). For example, the definitions:

```
(define signal counter
  (+ counter 1)
  (initial-value 0))

(define signal oscillator
  (not oscillator)
  (initial-value #f))
```

create signals that are defined in terms of themselves and whose data flow is therefore cyclic:

```
1

+                  counter

not

oscillator
```

The initial-value declaration tells the system that it should start with a value of 0 (or false for the oscillator) and then compute subsequent values for the signal using the previous value. You can include complicated cycles in a GRL problem in which \(a\) is defined in terms of \(b\), which is defined in terms of \(c\), which is defined in terms of \(a\), so long as one of those signals has an initial-value declaration. You can even have recursive groups, such as:

```
(define signal motor-command
  (let ((motor-command
        (+ motor-command
           (rt-vector rotate translate))))
    2)
  (initial-value (rt-vector 0 0)))
```

which would procedure a time-average of rotate and translate, or even:

```
(define signal silly-motor-command
  (rt-vector (translate-of silly-motor-command)
             (rotate-of silly-motor-command))
  (initial-value (rt-vector 1 0)))
```

which would alternately rotate and translate.

When defining complicated recurrences, it is useful to use the letrec special form, which is like let, but allows signals to be defined in terms of one another. For example, if we wanted a simplistic simulation of a robot’s emotions, we might say something like:
(define-signal emotions
  (letrec ((happiness (low-pass-filter (- pleasure fear) happiness-time-constant))
           (fear (low-pass-filter (- pain happiness) fear-time-constant)))
    (list happiness fear)))

which would make emotions be a list of two signals, happiness and fear, that are mutually inhibitory.

**Source signals**

GRL code compiles to some other language like C or Scheme. In order to interface your program to a sensor, you typically need to call some low-level procedure that has already been written in the target language. For example, to grab an image from the camera, you may need to call the C procedure `grab_image()`. The GRL source command lets you escape to the target language to specify arbitrary code to call to update a signal. For example, the definition:

```
(define-signal grayscale-frame
  (source (grab-image)))
```

Says that to update the signal `grayscale-frame`, it should call the low-level procedure `grab-image` with no arguments. When the compiler generates C code from this, it knows to change the Scheme-format call `(grab-image)` into the C-format call `grab_image()`.

**Declarations**

You can also annotate signals with declarations for the compiler. Declarations come after the value of the signal in the signal declaration. We’ve already seen the initial-value declaration in the context of recurrent signals. However, there are other declarations you can use as well.

**Type declarations**

In reality, the definition above for `grayscale-frame` wouldn’t work well, because it doesn’t tell the compiler what the type of `grayscale-frame` is. While the compiler can determine the types of constants and the results of calling built-in procedures, it can’t determine the types of source-signals itself. However, you can declare the types explicitly by including a type declaration in the signal declaration:

```
(define-signal grayscale-frame
  (source (grab-image))
  (type grayscale-image))
```

The GRL compiler understands the primitive types `integer`, `float`, `number` (meaning either integer or float), Boolean, groups, and vectors. The syntax for vector types is `(vector element-type length)`.

**Driving other signals**

Signals usually declare what their inputs are. That means that when you make a signal, you specify what its inputs should be. However, there are times when it’s more convenient to have a signal declare what its outputs should be and then leave it to the compiler to figure out who everybody’s inputs are. To tell the compiler that a given signal wants to send its value to some other signal, just use the `drives` declaration:

```
(define-signal stuck? (< center-distance 50)
  (drives status-window))
```

This tells the system that stuck? should be connected to the signal `status-window`, which we are assuming is some kind of a transducer signal that opens a window on the screen and shows the values of the various signals driving it (for example, there is a transducer in the `vutils` library for doing this called...
Only certain kinds of signals can accept their inputs from drives declarations. These signals are called *accumulators* and are made using the accumulate signal-procedure. *Accumulate* is just like *apply*, except it doesn’t take a list argument:

```scheme
(define-signal fear (accumulate +))
(define-signal startle-fear (* perceived-motion level-of-alertness))
(define-signal movie-fear (if just-saw-blair-witch? 100 0))
(define-signal ebola-fear (if just-read-crisis-in-the-hot-zone? 1000 0))
(define-signal current-events-fear (if just-saw-presidential-debate? 500 0))
```

This makes *fear* be a signal whose operator is + and whose inputs are determined at compile time. Whichever of the other signals get compiled along with *fear* then get linked up to *fear* as inputs. Thus, if you did `(try-signals fear ... some other signals ... startle-fear ebola-fear)`, you’d have a robot that could only be scared by being startled or reading about ebola. To take away its fear of ebola, just don’t compile the *ebola-fear* signal.

### Shared registers

Normal signals are “read-only” in the sense that you aren’t allowed to make assignments to them. Making an assignment to one wouldn’t make any sense, since the compiler automatically updates them on every cycle of the control loop. If you could write your own assignment statements for them, then the value that another signal saw would depend on whether the compiler’s automatic update or your assignment statement had executed most recently, which would lead to unpredictable behavior. However, there are often times when you want a signal that behaves like a normal C variable; you want it to keep its value until you do something to explicitly change it. While the state variables of transducers can be used this way, they’re hidden away inside of the signals and can’t be used as signals themselves. To a limited extent, you can make transducers, such as *latch* from the GRL library, whose only purpose is to provide a signal that acts somewhat like a normal variable. However, they can be inconvenient to use. More importantly, you can’t share one between two different transducers so that either one can assign it.

To solve this problem, GRL has special kinds of signals called *shared registers*. Shared registers are initialized once by the compiler and then left alone. They can be passed as inputs to normal signals, in which case the value seen by the value most recently assigned to that register. However, when passed as arguments to transducers, the transducers are allowed to modify them with `set!` commands just as with a normal state variable.

Shared registers are created with the procedures:

```scheme
(make-shared-register initial-value)
(make-shared-vector-register length initial-element-value)
```

### Plans and sequential execution

Finally, there are times when you just really want to write code that acts like normal sequential LISP or C code. The *plans* package exports the macros `plan` and `define-plan`, which compile a subset of Scheme into finite-state machines with normal sequential semantics. The major useful things you can do within plans are:
• Change the values of shared registers
• Call other plans, or special kinds of behaviors called actions
• Wait a specified period of time
• Wait for a specified event
• Call raw chunks of Scheme code

The major restriction on plans is that since they compile to finite-state machines, there is no stack for plan execution. Consequently, plans are not reentrant and therefore may not be recursive. Writing recursive or mutually recursive plans typically generates a deadlock condition in which a plan ends up waiting for itself to terminate before it may proceed with its own execution. For a further discussion of plans, see the GRL docs.

Other features

There are a number of other features that are outside the scope of this primer. The most notable are:

• Defining signal procedures as arbitrary scheme code
  Although, all the signal procedures we have shown here are written within the GRL language, it is also possible to write signal procedures that are implemented as raw Scheme code. They are called with the compiler data structures representing the signals and can then use arbitrary Scheme code to derive new signal data structures to return as the value of the signal procedure.

• Type analysis and overloading
  Signal procedures can check the types of their arguments and return different values as a result.

• Compile-time property lists for signals
  Declarations and signal procedures can store extra information in the signal data structures. This can be used to implement very sophisticated application-specific compilers for specialize robot languages. For example, the compiler for the plan language is implemented as a signal procedure which creates a custom transducer for each plan and then returns a signal which calls it. The shared registers which hold its arguments are stored in the property list of this signal so that they can be accessed when other plans which call it are compiled.

• Modalities
  Modalities are special signal procedures of one argument. They have two special properties. First, their definitions provide only default expansions for the signal procedure. When the modality is first called on the signal, the body of the modality is used to compute an expansion of the modality. However, it can later be replaced for that specific signal with an alternate expansion using the modes declaration. The other unusual property of modalities is that implementing them requires caching a pointer to their expansion within the signal itself. As a result, if you call the modality in three different places of your code on the same signal, you actually get back expansions that are eq? to one another (i.e. pointer-identical). This is sometimes used as a kind of poor-man’s common-subexpression elimination.

• Dynamically expanded transducers
  Transducers that take variable numbers of arguments
  You can create transducers that compute the Scheme code for their expansions dynamically based on the arguments passed to them. To do so, you specify a Scheme procedure to use as the expander of the transducer. It is called once at compile time for each signal that uses the transducer. It is passed the signal being compiled as an argument and it can then examine the inputs of the signal directly to determine what code to compile. The only times this has been used so far have been to implement the plan compiler and a few transducers used for GUI code that need to take variable numbers of arguments.