This document has two goals. One is to provide the user with activities to provoke thinking and discussion around the field of robotic art. The second is to provide the user with useful programming skills that will serve as a starting point for future explorations. The user should be prepared to conduct research in both fields outside of this document as well as working on a long-term project outside of these activities in order for the concepts introduced here to become clear.

Accompaniments to this document include the CricketLogo language reference (attached) and the Jackal programming environment.

Parts List:
1. Cricket
2. Interface Cricket
3. Light, temperature, touch sensors
4. Motors and motor cables
5. Distance, Clock, Tri-color LED, Big Motor, Stepper Motor, ??? Bus Devices
6. AC to DC power adapter (outside barrel positive)

Setting up:
1. Install Jackal.
2. Plug serial cable into computer serial port, Interface Cricket.
3. In Jackal, select Edit/Preferences and choose the serial port you are using.
4. Point the infra-red parts of the Interface Cricket and the Cricket towards each other.
5. In the Command Center in Jackal, type 'beep' and press return. The Cricket should beep.

Your first CricketLogo program:

Jackal has three windows- **Procedures**, **Command Center**, and **Run This**. The **Command Center** is for trying out single lines of instructions: type the instruction here and press return. Your programs will be written in the **Procedures** window. You can write as many procedures as you want in this window, each starting with 'to' and finishing with 'end', as shown below.

```
to test
  repeat 10 [note random 5 wait 2]
end
```

Now, in order for the Cricket to know what procedure to run when the white 'run' button is pushed, we must write the name of the procedure in the **Run This** window. Now, a few other points:

Primitives and Flow
Write a program to build a peanut butter and jelly sandwich. Here is part of mine (this isn't real, of course)*:*

```
to make-believe-peanut-butter-sandwich
  grab-knife
  open-jar (peanut)
  if(jar-open)[
     repeat 10 [ dip-knife  spread-substance]]
  close-jar (peanut)
  open-jar (jelly)
  if(jar-open)[
```

```
     repeat 10 [dip-knife spread-substance]]
  close-jar (jelly)
end

to grab-knife
??
??
end

to open-jar :n
??
??
end

??
```

What are the *primitives* (the native commands that the program is built out of, such as repeat)? What are the *procedures* (blocks of code starting with to ending with end)? For example, *make-believe-peanut-butter-sandwich* is one, the others are *grab-knife, open-jar.etc,* although they are not shown here. The *procedure make-believe-peanut-butter-sandwich* calls the other procedures, which do their thing and return. Are there collections of primitives that often get reused? What is the *flow* (how does the Cricket step through the program)? Does any process get *interrupted* by another? What are the *conditionals* (what conditions are checked, then acted upon)?

Apply the same questions to this CricketLogo programs. These *will* work on your Cricket, so plug in some motors, type the code in the **Procedures** window, type *dance* into the **Download This** window, and download them!

```
 to dance
  step-forward
  step-back
  repeat 2 [step-forward]
  repeat 2 [step-back]
end

to step-forward
  note 20 10
  thisway
  ab, onfor 5
end

to step-back
  note 30 10
  thatway
  ab, onfor 5
end
```

Bus
A bus is a communication method that allows computers to talk to other devices. In Crickets, using the bus is as simple as plugging in a device and remembering the right primitive that controls the device. For instance, plug in a digital display bus device and run the following program:

```
to test
  display 99
end
```

The primitive *display* tells the digital display to display a number, in this case 99.

Syntax
Computer languages have syntax just as written languages do. Syntax is a burden we just have to deal with. Luckily, CricketLogo has little.

Procedures must start with 'to' followed by a procedure name of your choosing and end with 'end'. Math operations must be spaced and brackets and parenthesis are used in many primitives.

Consider:

```
to blah
  if (sensora * 2 > 100) [beep]
end
```

Input/Output
Computers can have both inputs and outputs, such as the mouse or monitor that we are all familiar with. The Cricket has three 'primitive' outputs (motors, speaker, and infra-red.) and two inputs: (sensors and infra-red).

Sensora values are read with the primitives 'sensora' or sensorb'. Sensor values read from 0 to 255. Crickets can also use bus devices as an input or output device.

Please read through the attached CricketLogo reference to get a better understanding of the syntax and commands of CricketLogo. Through the following activities, take some time to lookup the commands used in the sample programs and also try to understand the structure of the programs. Analyze each sample program line-by-line to understand their flow. At anytime, break away from these activities to explore something in-depth.

Activity One: Actuation and Motors

Movement is the first quality we will explore that alludes to behavior. This activity focuses on animating a found object in ways that suggest life, intelligence, or irony it its movement. Please read the below choices, play with each, then build something, focusing on one quality of movement.

Built-in Motor Ports: The Cricket allows us to control the duration, direction, and speed of motors. Try this program after plugging in a motor to *Motor Port A* (see Cricket Diagram) on the Cricket:

```
to motor-test
   a, on             ;turn motor a on
   wait 10           ; wait one second
   a, rd             ;reverse the motor direction
   wait 10
   a, setpower 2     ;set the power lower, to 2 out of 8
   wait 10
   a, setpower 8     ;bump it back up to 8
   wait 10
end
```

Motor port B works the same way. Another motor command is *a, onfor*. This command turns on a motor for a given amount of time. However, the Cricket will move on to the next command while the motor is still on, waiting to be turned off. This is different than the *wait* command, which pauses the execution of the program for the duration of the wait. For example, the two below programs function very differently.

```
to motor-one         ;turns on motor a, quickly changes its
                     ;direction, after
  a, onfor 30        ;3 seconds shuts it off
  rd
end

to motor-two    ;turns on motor a, waits 3 seconds, changes
                ;its direction
  a, on
  wait 30
  rd
end
```

Gearing is still sometimes necessary, as are mechanisms for converting a motor's circular motion to linear motion. Fred Martin's *The Art of LEGO Design*

*(http://handyboard.com/techdocs/artoflego.pdf)*is a good introduction to prototyping mechanisms with LEGO.

Also, as a motor's turning rate is unreliable, we will also want to incorporate feedback to ensure precise positioning. This could be accomplished by having the moving object, or some part of the mechanism in motion, trip a touch sensor to stop the motion, change the speed or direction, or start a completely different behavior as well. Suppose we have a motor turning a lever of some sort, which at some point will run into a switch. Code for this situation might look like the following:

```
to motor-feedback
  a, on                        ;turn motor a on
  loop [  if(switcha)
             [a,off]
        ]                      ;continually check the state
                               ;of switch a- if
end                            ; pressed, turn off the motor
```

Servo Motors: Some motors have this feedback ability built into them- these motors are called 'servo' motors. The Cricket has a special board (bus device) that can control servo motors. Instead of turning them on or off (the servo motors used with this board cannot turn in full circles), we tell them what position to go to. The is accomplished with the command *turn-servo*, which we pass the servo motor number (labeled on the device) first, followed by the position we want it to go to. This position number must be experimented with in order to obtain the position you want. With the Servo Motor bus device connected to the Cricket and a servo motor plugged into it (black wire should be closest to edge), run the following program:

```
to servo-test
  turn-servo 1 20              ;turn servo 1 to position 20
  wait 5
  turn-servo 1 50
  wait 5
  turn-servo 1 90
  wait 5
  turn-servo 0
end
```

Stepper Motors: Stepper motors are another type of motor that can be quite useful. Stepper motors turn in angular steps, allowing the precise positioning of servo motors but can also turn all of the way around. The Stepper Motor bus device can control two steppers with the following commands:

```
a-step-speed :n          ;set the speed of stepper
                         ;a 0-100
```

```
a-step-forward          ;step forward forever
a-step-off                   ;stop stepping
a-step-brake                ;lock up motor so it won't
                          ;turn
a-step-back                  ;step backward forever
a-step-forwardfor :n         ;step forward of :n number
                             ;of steps  0-255
a-step-backwardfor :n        ;step backward for :n
                           ;number of steps  0-255
```

Note: Match up the color abbreviations on the Stepper Motor bus device when connecting a stepper motor.

Big Motors: One more actuation option is the use of big DC motors. While the Cricket can run DC motors on its own, you are limited to motors that run on less than 9V and draw less than .4 Amps. For people interested in controlling larger DC motors (needed for more power), the Big Motor bus device is needed. This device needs its own power supply, which needs have a connector with the outside barrel the positive supply, and inside barrel ground. The power supply can be anywhere from 5V to 40V. Once this and the motor are connected, the following commands can be used:

```
a-setpower :n
a-thisway
a-thatway
a-brake
a-off
b-setpower :n
b-thisway
b-thatway
b-brake
b-off
```

See the appendix for places to find cheap DC motors.

Artists and their work to view and discuss:

Marc Bohlen - http://www.contrib.andrew.cmu.edu/~bohlen/salt.htm
http://www.contrib.andrew.cmu.edu/~bohlen/alarm.htm
Gregory Barsamian - http://www.concentric.net/~Venial/sculptur.html
James Seawright - http://www.seawright.net/jamesseawright/motion.html

Activity Two: Sensing

Another interesting aspect of the Cricket is its ability to sense conditions of the physical world. In this activity, we will build an ambient display of an environmental factor. Four sensors plug directly into the sensor ports on the Cricket- they are light, temperature, touch (switch), and capacitive touch. These sensor values are accessed by the *sensora* or s*ensorb* command. The values range from 0 to 255. Try the following program after plugging in a sensor to Sensor Port A and a digital display.

```
to sensor-test
   loop[ display sensora]
end
```

Try out each sensor to get an idea of its range. For details of using the capacitive touch sensor, please see the appendix. Other sensors are bus devices, such as the distance sensor and the clap sensor. Plug in the optical distance sensor and try the following program:

```
to distance-test
  loop[ display ods-get-distance  ]
end
```

Strange results? Probably. These sensors have a point at which the number displayed will reverse- one nice way around this problem is to make them into a motion sensor with a simple algorithm. It looks like this:

```
global [dist1 dist2]

to motion-sensor
   loop[
     setdist1 ods-get-distance
     setdist2 ods-get-distance
     if (dist1 > 30)[
         if (((dist1 - dist2) > 10) or ((dist2 - dist1) >
10 )) [beep]
      ]
end
```

It uses *global variables* to store two different distance readings at slightly different times. It then checks to see if they are different by 10 or more. If so, motion has been 'detected' and it will beep. See the attached CricketLogo reference for the details of global variables.

Now, attach the clap sensor and a motor and try the following program:

```
to clap-test
  when[clap?][a, onfor 5]
```

```
end
```

Play with the dial to adjust the sensitivity of the device.

Now, design a way to display sensor information in a way suggestive of the information itself or another quality related to measured factor. Feel free to use the MIDI board here, as there is a long heritage of mapping sensor information to music. See the list of bus-device commands for the MIDI commands.

Here is one to get you started: It uses the Tri-color LED bus device and two sensors control the red and blue values of the LED.

```
to sensor-display
  loop[  cLED sensora 255 sensorb]
end
```

(The function cLED controls the Tri-color LED, with three arguments- the red, green, and blue value to display).

Artists and their work to view and discuss:
Amy Young - http://www.ylem.org/artists/ayoungs/index.html
Rania Ho –
http://www.ok-centrum.at/english/ausstellungen/cyberarts00/ho.html

Activity Three: Time

An interesting ability of the Cricket is its ability to keep track and respond to the date and time. This ability gives us the opportunity to create work whose behavior changes by date or time, evolves, or ages. This activity centers on a creation whose behavior changes during the course of the day. The clock needs to have constant power in order to keep the time, so make sure the backup battery is in place. The following functions are used to control the clock:

```
clock-init                    ;get the clock ready to write
to
set-time :hr :min             ;set the time
set-date :day :mth :year :dow ;set the day, month, year,
and day of week

get-day                  ;returns the day
get-mth
get-year
get-dow
get-hr
get-min
get-sec
```

For example:

```
to time-test
  clock-init
  set-time 12 30
  set-date 8 3 02 1
  do-something
end

to do-something
  loop[
     if ((get-time = 24) and (get-dow = 4))[a, onfor 20]
     ]
end
```

Notice that the first program, time-test, is used to initialize the clock. That program then *calls* another program, do-something, that constantly checks the time and reacts at midnight on Wednesdays.

Artists and their work to view and discuss:

Bruce Cannon - http://home.attbi.com/~brucecannon/

Ken Feingold - http://www.kenfeingold.com/docs/KF_01_2002.pdf

Activity Four: Communication

A unique ability of the Cricket is its ability to communicate to other Crickets via infra-red light. The primitive *send* sends a number (0-255), while the primitive *newir?* returns true if there has been a new IR reception, and the primitive *ir* returns the number received (and also sets *newer?* back to false). With only two Crickets talking back and forth, you do not have to be too concerned with protocol. One Cricket sends a number that represents a command to the other Cricket, which is expecting the number and knows what to do when it gets it. However, in order to take full advantage of this ability with many Crickets, we need to come up with a protocol to provide a bit more order.

Master/Slave

One possible protocol relies on having one Cricket being the 'master' and every other Cricket taking commands from it (the slave). A slight variation of this has each Cricket receive, execute, and pass on commands...acting at first like a slave and then like a master. For each of these protocols, a Cricket needs a unique identity and each command also needs a unique ID.

A sample program:

```
;setup variables to store ID, command
global [identity command1 command2 ir_val]

to setup
  setidentity 1          ;set identity to 1
  setcommand1 101   ;arbitrarily assign 101 to
                      ;command 1
  setcommand2 102
  receive_command   ;jump to new procedure
end

to receive_command
  when [newir?][         ;interupt on new ir
    if (ir = identity) [ ;are they talking to us?
       waituntil [newir?]          ;get the next ir
         setir_val ir                        ;save it
            if ir_val = command1 [do_this]
            if ir_val = command2 [do_that]
            send identity + 1           ;pass it on
            wait 5
            send ir_val
            ]
       ]
end
```

Who starts this process? What happens at the end? Can you get it to repeat itself?

We will now use the first protocol to create a collaborative work. Each person is responsible for creating a surprising or dramatic behavior for their particular piece that happens when their identity is received. Using the above example of code as the basis, everyone must choose an identity (no repeats!) and write two procedures (*do-this* and *do-that*) that will execute depending on whether the person before you sends you a 101 or a 102. The person with identity 1 will start the process. Make sure that the person after you in within the line of sight so that the IR signal will be received successfully.

Tips:

The primitive *newir?* reports a 'true' if there has been a new ir value come in since the last time you checked the value of *ir*, in a statement such as if(*ir* = 2). In other words, checking the value of *ir* clears the state of *newir?*. Store the value of 'ir' into a global if you want to uses it value more than once, as it might change if another Cricket is sending you more numbers.

Here is another master/slave example- figure out what it does!

global [identity ir_val]

```
to master
  send 63
  wait 5
  if (newir?) [
    setir_val ir
    repeat ir_val [beep wait 5]
   ]
end

to slave
when[newir?][
     if (ir = 63)[send identity]
      ]
setidentity 5
end
```

Artists and their work to view and discuss:

Simon Penny - http://www.telefonica.es/fat/vida2/alife/apenny.html

Eduardo Kac http://www.ekac.org/dialogical.html

Activity Five: Organism and Machine

In this activity, we will explore two modes of interaction: organism-like and machine-like. We will construct two separate works, one that attempts to mimic organism-like interactions and one that behaves in a machine-like way. This area is difficult as there are many complexities to grapple with. Artist Alan Rath, on the topic of behavioral sculpture, said, "what is 'interesting' behavior lies between doing nothing and randomness." Here is a very simple example of something machine-like that takes advantage of the computers ability to store and recall information quickly.

The Cricket can store 2500 points of data. There are 1440 minutes in a day- lets keep track of the light level once a minute all day long, then play it back in less than a minute!

```
to record-light
  erase 2500    ;erase all data
  repeat 1440 [record sensora wait 600] ;record light
values
end

to play-light
  resetdp
  repeat 1440 [cLED 0 0 recall]
end
```

(To try this out, use a light sensor in sensor port A and a Tri-Color LED for playback. Remove the *wait 600* and run *record-light*. Then run *play-light*.)

Artists and their work to view and discuss:

Jenn Hall - http://www.dowhile.org/physical/projects/acupuncture/index.html
Simon Penny –
http://www-art.cfa.cmu.edu/Penny/works/stupidrobot/stupidrobotcode.html
Edward Ihnatowicz -
http://members.lycos.co.uk/zivanovic/senster/index.htm#The%20Senster

Activity Six: Connecting to a Computer

The Serial Bus Device allows us to send and receive information with a computer. This allows us to send sensor values to control onscreen graphics or have the computer send commands to control the Cricket. Plug in the serial board to the Cricket, and connect it to the computer via a serial cable. In the below example we will use the Proce55ing graphics environment to receive sensor values from the Cricket and manipulate graphics based on these numbers. However, various other applications like Director (video) and Max MSP (sound) can receive and handle serial data as well.

Run this program on your Cricket:

```
to serial-test
  loop[
   send-serial sensora
   ]
end
```

Now, shut down CricketLogo open up Proce55ing, click run, and open up Sketchbook/Standard/SimpleSerialDemo.pde. Press the right triangle to run this program. If all is successful, the square onscreen should change its hue based on the sensor values.

Artists and their work to view and discuss:
Kenneth Rinaldo – http://www.ylem.org/artists/krinaldo/index.html

Odds and Ends

Where to get motors:
http://www.goldmine-elec.com/
http://www.allelectronics.com/
http://www.mpja.com/
http://www.sciplus.com/

Old washing machines, dryers, coffee grinders, disk drives, toys, computer fans...try running them from the Cricket at first, then from the Big Motor Bus Device. Look on the motor for voltage and current ratings printed on the motor. Also, the Cricket can control AC motors with the use of a relay. Buy one, and read about it. Use the motor port at full power (*setpower 8*) to flip the relay. Be careful with AC, and only work with someone who has done something similar before.

Look for 'gearhead' motors for slow, powerful DC performance.

A little about Analog and Digital:

At the simplest level, analog means a system that uses continuously variable voltages to relay information, while digital means that voltages are 'rounded off' to a high level and a low level (in the Cricket and a lot of other systems, this is 0V for low and 5V for high). As said before, the sensor ports on the Cricket measure an analog voltage. The Cricket then converts this analog voltage to a number value (the familiar 0-255). This process is called A/D conversion (for analog to digital).

Digital systems pass information by codes consisting of highs and lows. Remember high school math, when you learned about base-2 numbers? That is finally important! Numbers in base two (binary) are represented by 1's and 0's. This is the core of digital systems; numbers are passed around in base two, where the zero's are the low (0V) and the ones are the high (5V). The Cricket speaks digitally to the bus devices. The Cricket communication system is based on codes consisting of 8 ones or zeros. The ones and zeros are referred to as bits, and 8 bits is called a byte. Back in decimal representation, 8 bits (1 byte) are capable of representing a number between 0 and 255. Aha! The explanation of the mysterious 255!

Powering a Cricket from the wall socket:

To rid yourself of the need for batteries, you can build a power adapter for your Cricket. Go to Radio Shack and buy a 5V to 12V AC-to-DC adapter and a 9V battery clip (looks like the top of a 9V battery with a red and black leads coming from it. With the adaptor unplugged, cut the end off and strip the leads. With the leads separated as not to touch each other, plug the adaptor in and figure out with lead is positive and which is negative with a voltmeter. (Ask someone to help you here, especially if they own a voltmeter!).

Solder the negative end to the lead from the clip that is connected to the flanged side of the 9V connector. (It is probably the red one, although don't trust me here. Why red? The 9V battery clips are usually meant to clip a 9V into, not to take the place of a 9V! So, what we expect to be black (negative) is red (positive).) Solder the other lead of the adaptor to the lead connected to the smooth connector on the clip. Now, with the mulitmeter, make sure that the connector is the same as a 9V battery (flanged connector is negative, smooth one positive)! Be careful here, as a mixup here might kill your Cricket. With that confirmed, tape up your solder joints and plug it in!

Making your own sensors:

The Cricket can read a value between 0V and 5V. If you have a sensor that puts out an analog voltage between these values, you can plug it directly into the Cricket. The first slot up from the edge on the sensor port is the input. The second one is the ground. Plug ground from your sensor into ground of the sensor port (2^{nd} slot up), and signal from your sensor into the input (1^{st} slot up). However, many sensors are *resistive* sensors, which means that their resistance changes with environmental factors. The light sensor is an example of this, as its resistance changes with the amount of light hitting it (try it with an ohmmeter). For these types of sensors, we plug one end of the sensor into the 3^{rd} slot on the sensor port, which is held stead at 5V. The other end then goes into the input (the 1^{st} slot). The sensor port has a built in *voltage divider* that converts a variable resistance to a variable voltage when wired in this fashion. Try finding a *force sensitive resistor* or *bend sensor* at Radio Shack and using in this manner.

Things you know already:

This section provides analogies in CricketLogo to words you might come across in other programming languages:

- Procedures, functions, methods
These are segments of code that perform some function. We have been calling them procedures…anything that starts with a *to* and finishes in *end*.

-Variables
A variable allows for storage and manipulation of numbers in an abstract form. In CricketLogo, these are: *global [variable1 variable2]* where variable1 and variable2 are variables, of course.

-Recursion
Recursion is the method of calling a procedure from inside that same procedure.

```
to test
  beep
  test
end
```

Or better, but slightly more complicated:

```
global [variable1]

to test2
  recurse 1
end

to recurse :n
  setvariable1 :n + 1
  display variable1
recurse variable1
end
```

-Arguments

A procedure can take one or more arguments as inputs or output one argument. Below, :n is the argument.

```
to beep-thismany :n
  repeat :n [beep wait 2]
end
```

To use this procedure, we write:

```
to test
  beep-thismany 12
end
```

- Conditionals or Condition checking

Checks a Boolean condition. Allows events to be conditional.

```
to check1
  loop [if (switcha) [beep]]
end
```

or:

```
to check2
  loop[if (sensora > 200)[beep]]
end
```

- Commenting and Reuse of Code

Commenting (text that isn't part of the code, but explains what a piece does) allows for other people to understand what your code does. Compartmentalizing code and using

arguments helps make your code reusable. In CricketLogo, comments are started with the ';'.

- Threads/Multitasking

Multitasking allows the computer to perform several tasks simultaneously. In CricketLogo, our only multitasking ability is the *when* command.

```
to thread      ;interrupts the loop to beep when condition
is true
  when [switcha][beep]
  loop[a, on wait 4 a, off wait 5 rd]
end
```

Useful books for going further in electronics, robotics, and sensors:

Robotic Explorations: A Hands-on Introduction to Engineering
Fred Martin
Addison-Wesley
ISBN: 0130895687

Electronic Circuit Guidebook: Sensors
Joseph J. Carr
PROMT Publications
ISBN: 070610981

McGraw-Hill Benchtop Electronics Handbook
Victor Veley
McGraw-HIll
ISBN: 0070674965

TAB Electronics Guide to Understanding Electricity and Electronics
G. Randy Stone
Tab Books
ISBN: 0070582165

The Art of Electronics
Paul Horowitz, T. Hayes
Cambridge University Press
ISBN: 0521377099

Handbook of Modern Sensors
Jacob Fraden
Springer
ISBN:1563965380

Interesting reads regarding interactive sculpture:

Beyond Modern Sculpture
Burnham, Jack
George Braziller, New York

Art of the Electronic Age
Popper, Frank
Thames and Hudson
ISBN 0-500-27918-7