

Physics 219 – Fall, 2005
LabNotes 8

A Reintroduction to LogoChip Logo

Table of Contents

Wednesday, November 16	1
LogoChip Logo Syntax.....	1
Common syntax errors	3
PicoBlocks: A graphical view of Logo syntax	4
Logo and Procedural Abstraction.....	6
LogoChip Logo by Example.....	7
Build a test bed.....	7
Follow the bouncing control dot.....	9
Edge-triggered vs. level-triggered logic.....	9
Programming Challenges	10
Concurrency.....	10

Wednesday, November 16

While you’ve already written quite a few programs for LogoChip, you’ve done this without much formal instruction. Now seems like a good time to step back and spend a day reintroducing the LogoChip Logo language and highlighting its main features and some of the “big ideas” that you’re likely to encounter as you write programs.

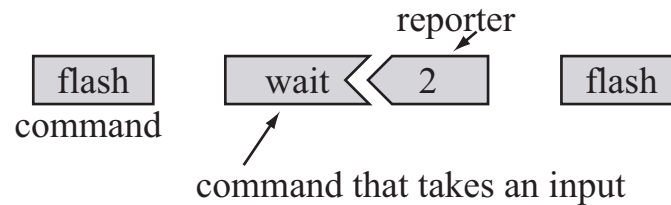
LogoChip Logo Syntax

The word **syntax** refers to the grammatical structure of a computer language. Logo has a particularly simple syntax, as far as programming languages go. Consider the following line of code:

```
flash wait 2 flash
```

First some terminology. **Flash** and **wait** are examples of Logo **procedures**, parts of a program that perform some specialized task. They are both **primitive** procedures, because Logo knows how to do them with any guidance from you. They are “built in” to the language. **Flash** is an example of what we will refer to as a **command**. A command does not output a value, but rather has some *effect*,

that causes something to happen, like flashing the LED. **wait** is an example of a command that requires an **input**. We can depict the situation graphically, as in the cartoon below:



A **reporter** is a procedure that computes a value and **outputs** it. A **number** like “2” is a simple example of a reporter. Another example is the “+” procedure, which that takes two inputs (one on either side) and outputs the sum.

```
print 5 + 2
```

(In this example `print` is a command that takes one input.) In LogoChip Logo all reporters output **16-bit integers** that have value from -32768 through +32767.

Logo cares a lot about **spaces**. It uses spaces to determine where one command or reporter begins and the next one ends. For example typing

```
print 5+2
```

will lead to an “I don’t know how to 5+2” error message.

Square brackets enclose a **list** of commands:

```
repeat 5 [flash wait 5]
```

Booleans are expressions that are either “true” or “false”. A control structure like `if` makes use of a Boolean input to decide whether to execute a list of commands

```
setn random if (n > 10000) [flash]
```

In LogoChip Logo any non-zero integer is evaluated as “true” while zero evaluates as “false”:

```
if 57 [flash] ;the LED flashes
```

```
if (10 - 10) [flash] ;the LED doesn't flash
```

Parentheses are used to determine the “order of evaluation”:

```
print 5 + 2 * 3      ; prints 11
```

```
print (5 + 2) * 3   ; prints 21
```

Common syntax errors

- Missing spaces:

```
wait10
```

Error: wait10 is undefined

```
wait 2*5
```

Error: 2*5 is undefined

- Too few/too many inputs:

```
setbit 0
```

Error: not enough inputs to setbit

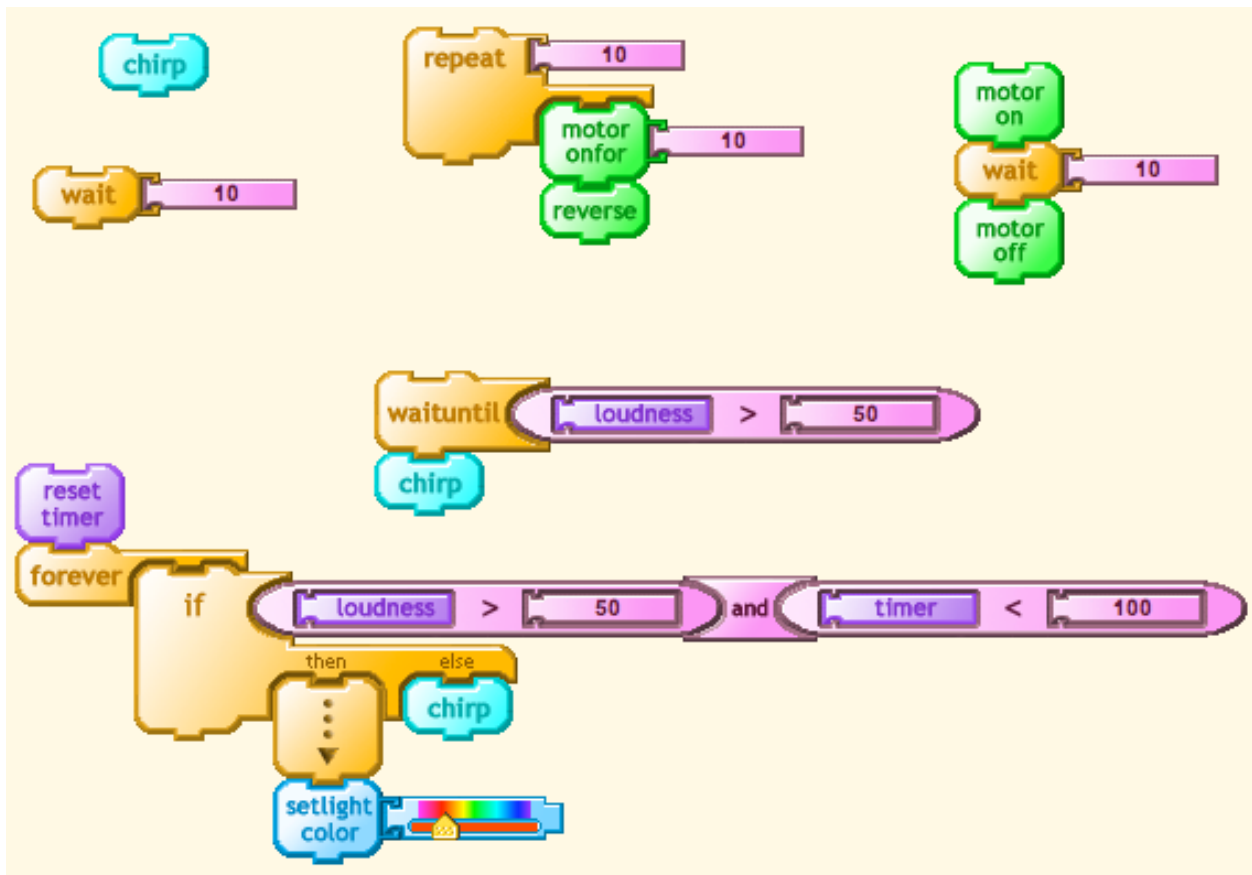
- Command where reporter expected:

```
setbit 0 wait 10
```

Error: wait doesn't output

PicoBlocks: A graphical view of Logo syntax

One way to get an overview of the LogoChip Logo syntax is to spend some time looking at *PicoBlocks*, the programming language used by the soon-to-be-released *PicoCrickets*. *PicoBlocks* is a graphical programming language whose underlying syntax is virtually identical to LogoChip Logo. Probably the most important “design criterion” for *PicoBlocks* was that it had to be both a powerful programming language that enabled users to eventually do all sorts of sophisticated things, and at the same time be very easy to start using. Even with the relatively simple syntax of a language like Logo, our previous experience with text-based versions of Logo showed us that issues related to brackets and spaces and parentheses and misspellings led to significant “entry barriers” for novices. *PicoBlocks* addresses this challenge using a building blocks metaphor in which the shapes of the puzzle-piece-like blocks give a great deal of syntactical guidance. In *PicoBlocks* it’s virtually impossible to make many of the syntax errors that are common in the text-based version of the language.



Commands:



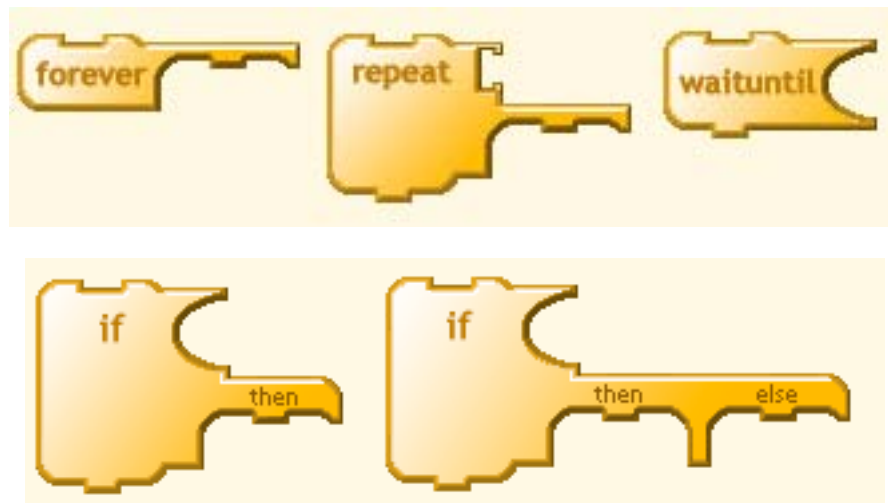
Reporters:



Command that takes an input:



Flow Control:



Booleans:



While easy to learn, there are some limitations to the graphical approach. Long programs begin to get unwieldy and some very useful programming constructs are not possible in the graphical language. For this reason, there is a one-to-one correspondence between a *PicoBlocks* program constructed graphically and a text

version of the program. A text-based environment, quite similar to LogoChip Logo, is located on the “flip-side” of the screen. This makes it easy for more advanced users to make a transition to writing more sophisticated programs using the text-based part of the language.

Logo and Procedural Abstraction

What if you want to do something really complicated? A big idea in programming is that want to be able to extend the vocabulary of whatever language you have by creating your own procedures in a text file:

```
to five-flashes
  repeat 5 [flash wait 2]
end
```

After pressing the **download button**, the LogoChip understands this new command. (We can choose any name we like for new the procedure name, as long as it doesn't conflict with existing names.)

When building new vocabulary, we can incorporate other vocabulary we've built. This is very powerful for making complex programs.

Very often, want to write commands that are **parameterized** over certain values. E.g., you might want to control the number of times the light flashes:

```
to flashes :times      ; colon indicates argument -
                        there is no space after the
                        colon
  repeat :times [flash wait 2]
end
```

You can now invoke `flashes` with different numbers:

```
flashes 3
flashes 5
```

You can use multiple parameters:

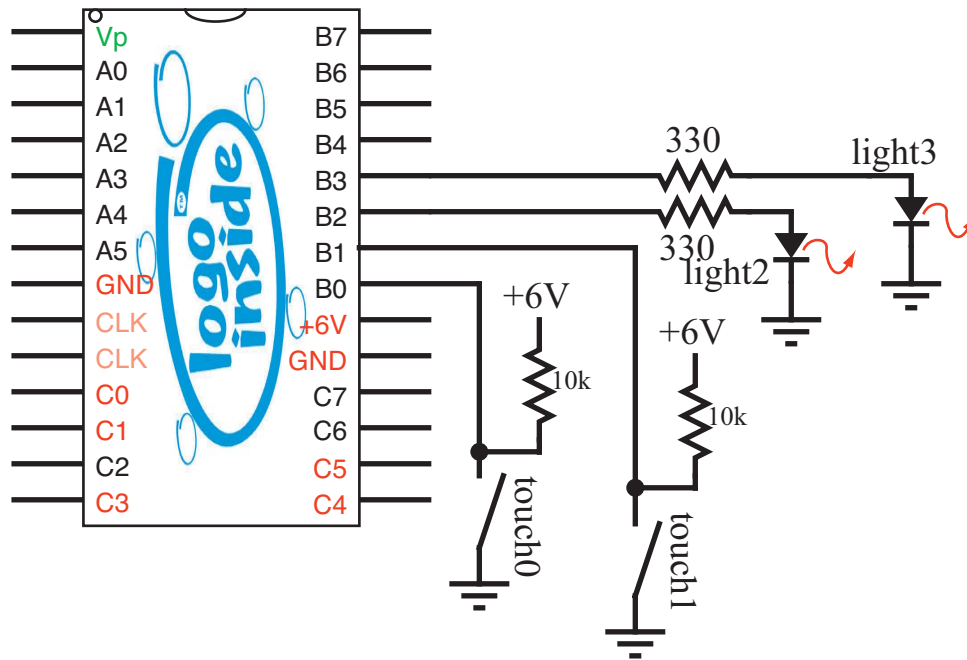
```
to flash2 :times :interval
  repeat :times [flash wait :interval]
end
```

(Note: The names chosen for parameter names are “local” to the procedure. They won’t conflict with names in other procedures.)

LogoChip Logo by Example

Build a test bed

- Wire up two touch sensors and two LEDs to portb, as shown in the figure below.



- Define the following procedures:

```
to initialize
clearbit 2 portb-ddr
clearbit 3 portb-ddr
end
```

```
to touch? :num
output not (testbit :num portb)
end
```

```
to on :num
  setbit :num portb
  mwait 25      ; grab control for a bit to avoid
                  switch bounce problems
end
```

```
to off :num
  clearbit :num portb
  mwait 25      ; grab control for a bit to avoid
                  switch bounce problems
end
```

```
to toggle :num
  togglebit :num portb
  mwait 25      ; grab control for a bit to avoid
                  switch bounce problems
end
```

```
to onfor :num :time
  on :num
  wait :time
  off :num
end
```

```
to toggle :num
  togglebit :num portb
  mwait 25      ; grab control for a bit to avoid
                  switch bounce problems
end
```

- Try typing the following commands

loop [print touch 0 wait 5] Prints 1 (true) when touch sensor #0 is pressed, 0 (false) when not pressed.

on 2 waituntil [touch? 0] off 2 Turns Light2 on; pressing touch sensor #0 turns it off.

```
loop [waituntil [touch? 0] onfor 2 10]
```

Turns Light2 on for a second every time touch sensor #0 is pressed. Light stays on if switch is held down.

Follow the bouncing control dot

- What happens when you press the button?

```
loop [if touch? 0 [toggle 2]]
```

- Type long commands like the following in the Command Center *without* a line return!
- Touch sensors #0 and #1 turn on Light #2 and Light #3 in alternation. Switch ignored when (1) motor on (2) not its "turn":

```
loop [waituntil [touch? 0] onfor 2 10 waituntil
      [touch? 1] onfor 3 10]
```

Touch? 0 turns Light 2 on, touch? 1 turns it off:

```
loop [waituntil [touch? 0] on 2 waituntil [touch? 1]
      off]
```

The following does *not* toggle Light 2 on and off. Why?

```
loop [waituntil [touch? 0] on 2 waituntil [touch? 0]
      off 2]
```

Edge-triggered vs. level-triggered logic

- Touch? 0 toggles Light 2 on and off. This is an example of **edge-triggered action**:

```
loop [waituntil [not touch? 0] waituntil [touch? 0]
      toggle 2]
```

- Touch? 0 turns on Light 2, touch? 1 turns on Light 3, any order. Switch

ignored when a light is on:

```
loop [if touch? 0 [onfor 2 10] if touch? 1 [onfor 3
                                     10]]
```

- Light 2 is on when touch? 0 is pressed and off otherwise. This is an example of **level-triggered action**:

```
loop [ifelse touch? 0 [on 2] [off 2]]
```

- Predict, and then test, the behavior of the following commands:

```
on 2 if touch? 0 [toggle 2 ]
```

```
on 2 waituntil [touch? 0] toggle 2
```

```
on 2 loop [if touch? 0 [toggle 2 ]]
```

```
on 2 loop [waituntil [touch? 0] toggle 2]
```

```
on 2 loop [waituntil [not touch? 0] waituntil [touch?
                                               0] toggle 2]
```

Programming Challenges

- Write commands to implement the following behaviors:
 - 1) Light 2 is on when touch sensor #0 is pressed and off otherwise; Light 3 is on when sensor #1 is pressed and off otherwise.
 - 2) Touch? 0 turns Light 2 on and Light 3 off, touch? 1 turns Light 2 off and Light 3 on (in any order)
 - 3) Only one of Light 2 and Light 3 is on. Which one is on changes every time touch? 0 is pressed.

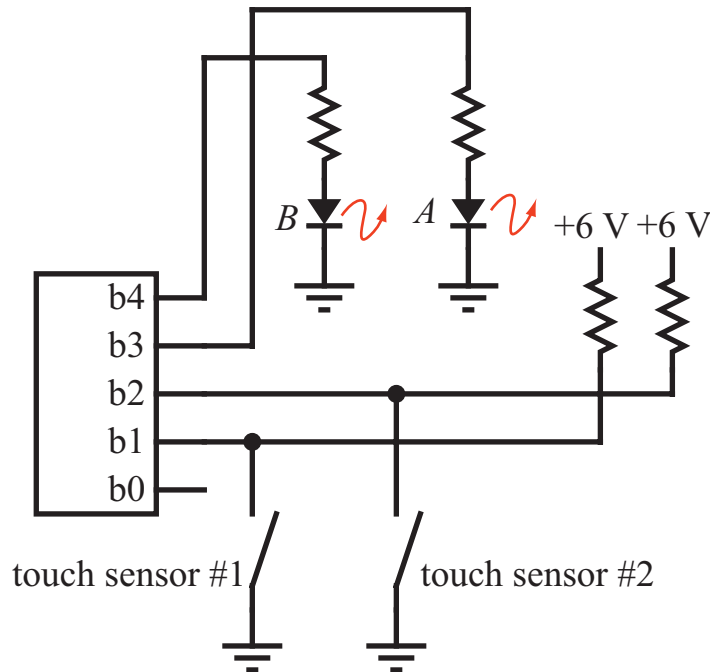
Concurrency

Concurrency refers to a language's ability to do many things at once. You can think of it as a language having the ability to generate multiple control dots.

Bad News: LogoChip Logo has no concurrency in this sense. There is only one

control dot. (Not to get too defensive about it, but after all it *is* a \$5 computer.) That makes problems like the following one from problem set #3 tricky:

Problem: Touch sensor #1 is wired to pin b1 of a LogoChip, touch sensor #2 is wired to pin b3 b2 and LEDs A and B are wired to pins b3 and b4, as shown in the figure below.



Write a Logo program so that causes LED A to “toggle” (turn on if it is off and turn off if it is on) whenever touch sensor #1 is pressed and LED B to toggle whenever a touch sensor #2 is pressed. (**Hint:** You will need to use “global variables” to solve this.)

The problem is that you need to *simultaneously* look for edges on two the two switches. Using waituntil won’t work. As the hint suggests, you’ll need to use variables¹ the remember “past state” and check for changes in state.

¹ In LogoChip Logo there are two built-in global variables called m and n. The commands `setm` and `setn` are used to set the value of these variables. For example

```
setn 5
```

will “set the value of n to 5” by which we mean that n will report a value of 5.

Additional global variables are created by including the `global [variable-list]` directive along with the procedures definitions. *E.g.*,

Good news: The LogoChip has lots of built-in special “hardware” features that really help you perform various task concurrently with the program that is running. You’ve already seen the built-in pulse width modulation features. In addition the LogoChip has a number of built-in **counters** especially designed for counting “edges” that occur on certain input pins and also a number of internal **timers** that can keep track of the elapsed time between events..

```
global [foo bar]
```

creates two additional globals, named `foo` and `bar`. Additionally, two global-setting primitives are created: `setfoo` and `setbar`. Thus, after the global directive is interpreted, one can say

```
setfoo 3
```

to set the value of `foo` to 3, and

```
setfoo foo + 1
```

to increment the value of `foo`.