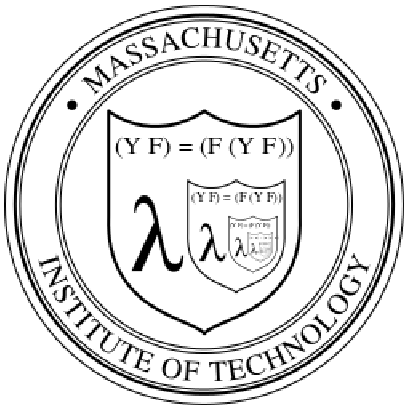
adafruit learning system

Alternative languages for programming the SAMD51 boards

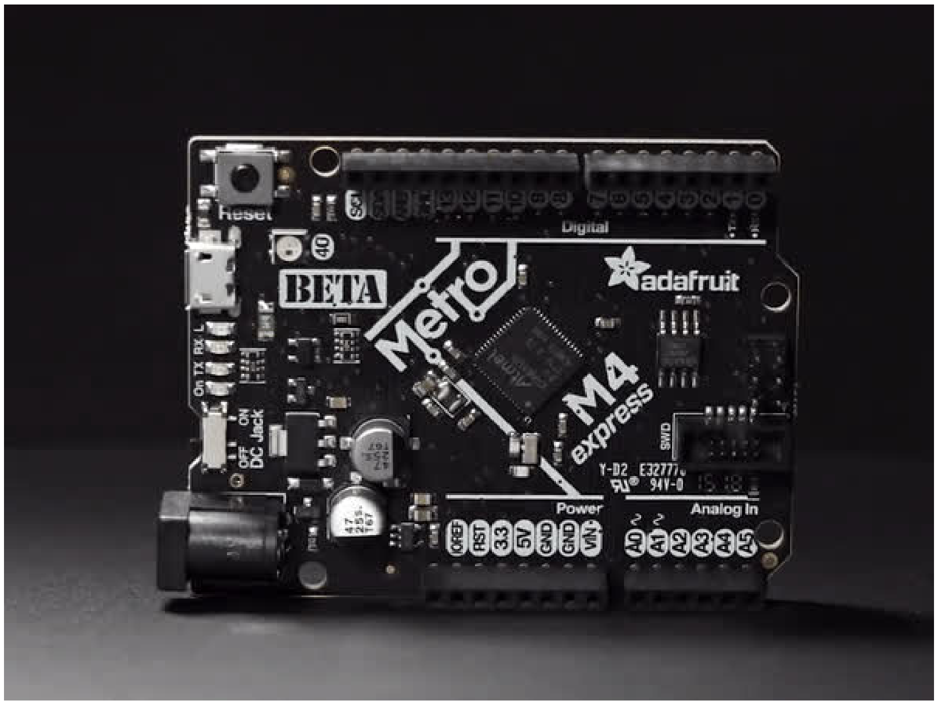
Created by Dave Astels



Last updated on 2018-09-16 12:30:34 AM UTC

Guide Contents

Overview



This guide is going to bring something "new" to the table. Adafruit's SAMD21 (aka M0) and SAMD51 (aka M4) boards provide impressive support for various programming options, some of which haven't really been a viable option on microcontrollers. Of course, there's C and C++. That's available for pretty much any microcontroller ever. The '21 and especially the '51 throw open the gates to higher level languages such as Makecode/Javascript and CircuitPython.

Arduino/C/C++ is the classic way for makers to write code for microcontrollers. It's complex, demanding, and requires fine attention to detail. It can also be intimidating to new makers.

Makecode is a wonderful way for beginners to get started making microcontroller based things quickly and easily. It lets you code without getting bogged down in syntax, indentation, etc.

CircuitPython is the new kid on the block. It's a fork of MicroPython primarily for Adafruit's SAMD series of boards, extending MicroPython to make it even easier to get up and running. MircoPython has been around since May 2014, but CircuitPython's first (pre-)release was in Aug 2016, making it quite new. The SAMD boards have the memory and speed to make Python a viable alternative.

So with all this, why look at other alternatives? Partly because it can be fun to learn new ways to do things. But also, the more tools you have available, the more versatile you can be.

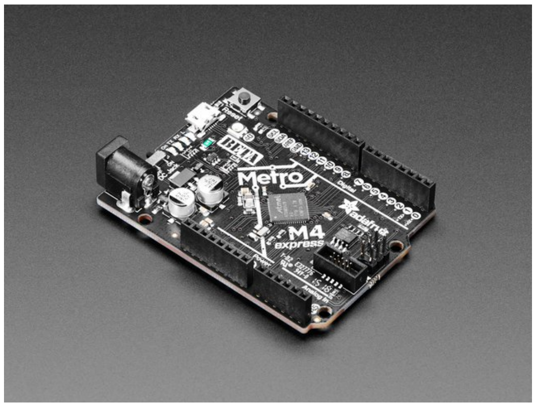
This guide will have a look at a couple languages that can be used for programming microcontrollers that haven't really been mainstream in the maker world yet: Forth and Lisp.

While we'll mainly look at using these with the latest M4 boards from Adafruit (Metro, ItsyBitsy, and Feather), there are Forth and Lisp implementations available for boards all the way down to the venerable Arduino UNO. Of course, the more capable microcontrollers boards (especially those using the SAMD51) will be able to do more, and do it faster.

Forth and Lisp are very, very different languages. Yet they have some things in common: they look nothing like C, Python, Ruby, JavaScript, or probably most programming languages you've used. Another thing they share is that they've both been around for a long time. Lisp dates back to 1958 and the beginning of the AI (Artificial Intelligence) Lab at MIT. In fact the fellow who conceived of and designed Lisp, John McCarthy, also came up with the phrase "artificial intelligence". Forth is slightly newer, dating back to the late 1960s work by Charles Moore.

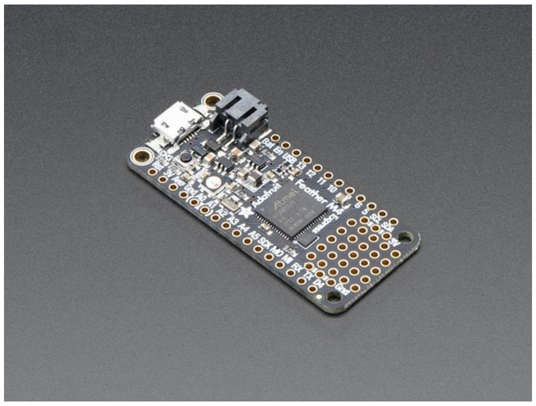
Learning Lisp or Forth isn't just learning a slightly different syntax; it's learning a new way of programming. If that idea excites you, this guide will give a very brief introduction to each language, an implementation of each that will run easily on the new M4 boards, and pointers to more information. Neither implementation has the level of polish that CircuitPython has, or that of more mature workstation implementations. That said, they are gradually improving.

Of course there are implementations available for more capable computers as well, especially Linux-based ones. So if Raspberry Pi is your jam, you have a wealth of alternatives available for both of these languages.



Adafruit Metro M4 feat. Microchip ATSAMD51

$27.50
IN STOCK
ADD TO CART



Adafruit Feather M4 Express - Featuring ATSAMD51

$22.95
IN STOCK
ADD TO CART

Your browser does not support the video tag.

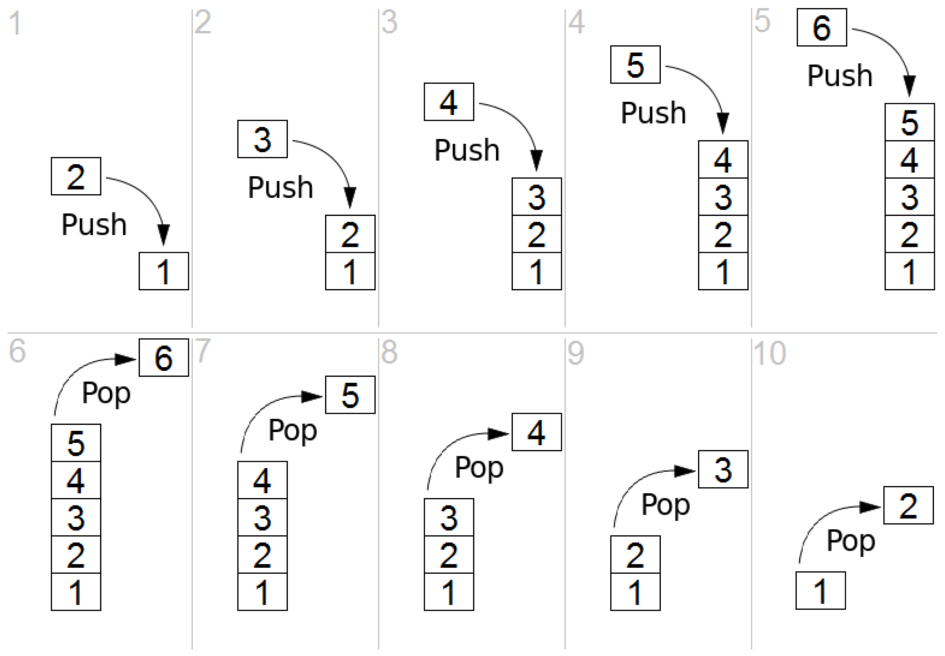
Adafruit ItsyBitsy M4 Express featuring ATSAMD51

$14.95
IN STOCK
ADD TO CART

The key image of this guide is the logo of MIT/GNU Scheme, the canonical implementation of the Scheme dialect of

Lisp. It is shared under the terms of the GNU Free Documentation License, Version 1.2. It was chosen here because it portrays the recursive nature of programming in Lisp.

Forth



The Forth language is quite different for most others, certainly from C or Python. The syntax is different. The architecture is different. How arguments and results are passed around is different.

Forth has been around since the late 1960s has been popular for embedded programming, and is a logical choice for makers and IoT projects: it thrives in environments where memory is limited and efficiency is important. This makes it perfect for use with microcontrollers.

There are a few Forth implementations available for microcontroller boards, but the one we're going to look at has explicit support for Adafruit's latest SAMD51 boards (experimental but seems to work). It's a version of ainsuForth, specifically (https://github.com/wa1tnr/ainsuForth-gen-exp-m4 (https://adafru.it/BLp). Github user `wa1tnr` is none other than `@nis` from the Adafruit Discord!

Installing

An advantage of `ainsu` is that it's packaged as an Arduino project; once you clone/download the repo, you just load it into the Arduino IDE then compile and flash onto your board.

If you're not familiar with the Arduino IDE and it's use, we have a guide for that to get you up and running (https://adafru.it/BLK). That will work with any of the SAMD boards.

Download the code from Github (https://adafru.it/C2u). Inside the **ainsuForth-gen-exp-m4-master** directory you get when you unzip the file you just downloaded will be a file named **ainsuForth-gen-exp-m4.ino**. Start up your Arduino IDE and open that file. Select your board and port as usual. Build and upload to your board. You will need to place your M4 board into bootloader mode by double pressing reset in order for the uploader to see it.

Connect via the Arduino serial terminal, screen or whatever way you use to connect to the Serial I/O on your board. When you do you'll be greeted by the REPL. Forth will output a banner with some information and then wait for you to type code. Try `1 2 + .` followed by the return/enter key. In response you should see `3 ok`

```
         ainsuForth - 2018 - wa1tnr    words: 81
         YAFFA - Yet Another Forth For Arduino,
         Copyright (C) 2012 Stuart Wood
1 2 + . 3  ok
```

Stack Machines

Forth is a stack based language. That means that passing arguments to functions and getting results from them is done using Forth's stack instead of sending arguments in the function call and having a returned result.

A Stack Refresher

If you've been programming for a while, you may be familiar with stacks. In the case you're not, let's have a quick look at this very useful data structure.

A stack is a data structure you can put things into and take things out of. Its most interesting feature is that things come out in the reverse order of how they were put in. The act of putting something into a stack is called *pushing* and removing is *popping*.

For example, if `1`, `2`, and `3` are pushed into a stack in that order, the thing that will be popped first is the `3`. The next pop will return the `2`. And so on. At any point the most recently pushed item is available *at the top of the stack*.

You can visualize a stack as a pile of items. When you push you are placing an item on the top of the pile, and when you pop you are taking the item from the top of the pile. You generally can't see anything in the pile except for the thing on top (i.e. the last thing that was pushed). Forth does provide ways to peek deeper into the stack or make a copy is items deeper in the stack (which are then pushed onto the top).

Trying to access the top of an empty stack or pop from it results in an error. In some cases there is a maximum size to a stack; pushing onto a full stack will be an error.

Syntax

When they appear in code, simple data items like numbers get pushed onto the stack. Names of functions (called *words* in Forth) cause the named function to be executed. So the general pattern is to put arguments on the stack, then use a word. For example, to add `2` and `3`:

```
2 3 +
```

This pushes `2`, then pushes `3`, then executes the function named `+`. The `+` function pops two numbers, adds them, and pushes the result onto the stack.

If a word has a result, as in this case, it is left on the stack when the word finishes executing. Because of this, you don't have to store results in a variable to be passed into another function; it just sits on the stack waiting to be used. For example `(2 + 3) * 4` could be coded as:

```
2 3 + 4 *
```

The result of `2 3 +` (which is `5`), is left on the stack, `4` is then pushed onto the stack and the `*` function pops two arguments from the stack (`4` then `5`), multiplies them and pushes the result (`20`) back onto the stack.

The `.` word will pop the top item from the stack and display it . If you type some code in the REPL that leaves a result on the stack you'll need to use `.` to see it.

Another word that's handy, particularly used with `.` is `dup`. It makes a copy of the top item on the stack, and pushes it.

Combining the two, you can use `dup .` to see the item on the top of the stack without changing it.

Everything in Forth, data items and words alike, are separated by whitespace. Unlike Python, indentation is irrelevant and whitespace serves only to separate words.

Documentation

Because of the stack architecture of Forth, words are documented a bit differently. If you look at docs, you'll see something like

```
+ ( n1 n2 -- sum )
Adds.
```

The part in parentheses defines the inputs and outputs of the function. things to the left of the `--` are parameters that will be popped from the stack by the word. Things to the right are the results. Words can have any number of results. They're simply left on the stack when the word has done its job.

Hardware Control Words

The word `pinMode` sets the mode (input/output) of a pin. To set pin 13 (connected to the onboard LED) to output you would use

```
1 13 pinMode
```

This is documented as:

```
pinMode (mode pin -- )
Set pin to mode.
```

Similarly, there is `pinWrite`:

```
pinWrite (value pin -- )
Write value to pin.
```

So to turn on the onboard light we could (after setting the mode) use:

```
1 13 pinWrite
```

To turn it off we can use:

```
0 13 pinWrite
```

To abstract (as we love to do as programmers) and make our code easier to read, we can write some words for those operations. To write a word, we start with a colon (`:`), followed by the name of the new word, followed by the code, and finally terminated by a semicolon (`;`).

```
: ledOn  1 13 pinWrite ;
: ledOff 0 13 pinWrite ;
```

With those in place we can simply use the `ledOn` and `ledOff` words to control the LED.

There's a `delay` word, of course, that echos the functionallity of the Arduino C equivalent:

```
delay (millis -- )
Sleep for millis milliseconds.
```

Blink

Now we can write code for a blink word that will blink the led some number of times:

```
\ blink (count --)

: blink
0 do
  ledOn
  300 delay
  ledOff
  400 delay
loop ;
```

To blink 5 times, you would write `5 blink` .

From above, the lines

```
ledOn
300 delay
ledOff
400 delay
```

should be fairly understandable: turn the led on, sleep 300mS, turn it off, and sleep 400mS.

The `do` and `loop` words require a bit more explanation, and should shed a little more light on how Forth works.

The doc for `do` is

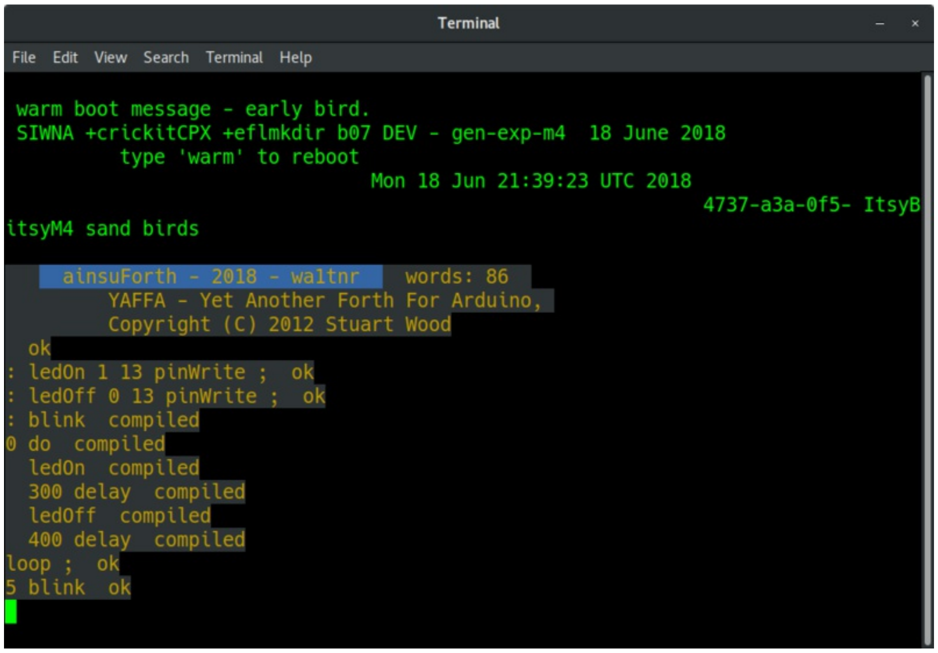
```
do (limit initial -- )
```

The body (everything between `do` and `loop`) is executed for each value from `initial` up to (but not including) `limit` . So, if initial is 0, it behaves just like Python's

```
for _ in range(limit):
```

The loop word increments the current index and compares it to limit. If it's still less, the loop repeats. If it's equal, the loop ends and the program continues with the code following loop.

A key to this example, and generally understanding how Forth works, is to notice that do pops two parameters from the stack. In our blink word, we pushed a 0 then used do. Where did it's other parameter come from? We put it on the stack before calling blink in 5 blink. So when do pops its parameters, there's a 0 on the top of the stack, and a 5 just below it.



```
 warm boot message - early bird.
 SIWNA +crickitCPX +eflmkdir b07 DEV - gen-exp-m4  18 June 2018
          type 'warm' to reboot
                                Mon 18 Jun 21:39:23 UTC 2018
                                                             4737-a3a-0f5- ItsyB
itsyM4 sand birds

   ainsuForth - 2018 - wa1tnr    words: 86
        YAFFA - Yet Another Forth For Arduino,
        Copyright (C) 2012 Stuart Wood
  ok
: ledOn 1 13 pinWrite ;  ok
: ledOff 0 13 pinWrite ;  ok
: blink  compiled
0 do  compiled
  ledOn  compiled
  300 delay  compiled
  ledOff  compiled
  400 delay  compiled
loop ;  ok
5 blink  ok
```

Limitations

ainsu Forth is very much a work in progress. It doesn't yet have support for I2C, SPI, or other interfaces you would expect in a system for programming microcontrollers. That said, it's a good start and all the language mechanisms are in place. It's opensource so anyone can get involved and extend it.

It would benefit from having a way to use the Express boards' SPI flash filesystem the way CircuitPython does. As of this writing, it's awkward getting code into it: pasting into a serial session.

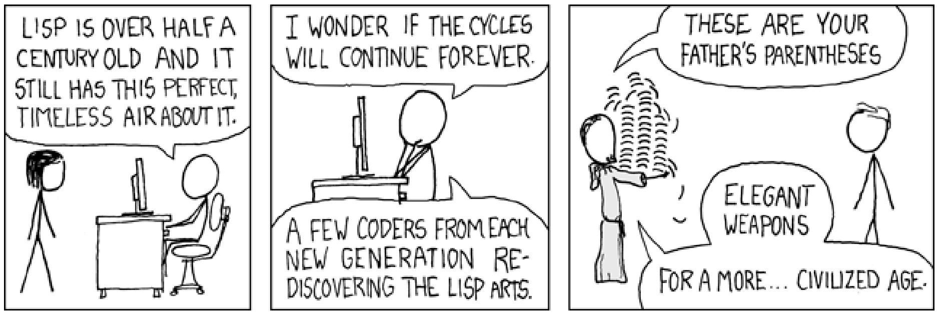
Resources

That should give you a little taste of what Forth is like. If it leaves you curious and wanting to know more, here are some resources:

- Starting Forth (https://adafru.it/BLq) is one of the classic Forth introduction and tutorial. PDF and online versions are freely available.
- https://github.com/wa1tnr/ainsuForth-gen-exp-m4 (https://adafru.it/BLp) A nice implementation supporting the SAMD51 chip used in Adafruit's M4 boards.
- FORTH, Inc. (https://adafru.it/BLr) A long time Forth consultancy. Their site contains a wealth of information.

The SAMD chips are relatively new and there are existing ARM Cortext-M0 and M4 versions of FORTH, so expect more alternatives to be available before much longer.

Lisp


LISP IS OVER HALF A CENTURY OLD AND IT STILL HAS THIS PERFECT, TIMELESS AIR ABOUT IT.
I WONDER IF THE CYCLES WILL CONTINUE FOREVER.
A FEW CODERS FROM EACH NEW GENERATION RE-DISCOVERING THE LISP ARTS.
THESE ARE YOUR FATHER'S PARENTHESES
ELEGANT WEAPONS
FOR A MORE... CIVILIZED AGE.

LISP was one of the earilest programming languages, dating back to 1958 (it recently had its 60 anniversary as I write this). That makes it one of the oldest programming languages. Of languages that still see use today, only Fortran is older. It also has the distinction of being a direct or indirect influence on most of the languages in heavy use today. I'll bet you use all sorts of language features that were first implemented in LISP. Examples are conditional statements (i.e. if, else, ...), REPLs, and automatic garbage collection.

For the purposes of this guide, we'll be using ulisp (https://adafru.it/BLu) which has versions that run on MCUs all the way down to the ATMega328. Of course the 32 bit ARM versions run faster, and let you write longer programs. As of version 2.4 uLisp-arm officially supports Adafruit's M4 line of boards.

Installing

An advantage of ulisp is that it's packaged as an Arduino project; once you clone/download the repo, you just load it into the Arduino IDE then compile and flash onto your board.

If you're not familiar with the Arduino IDE and it's use, we have a guide for that to get you up and running (https://adafru.it/BLK). That will work with any of the SAMD boards.

Download the code from Github (https://adafru.it/Cwx). Inside the ulisp-arm-master directory you get from unzipping the file you just downloaded will be a file named ulisp-arm.ino. Start up your Arduino IDE and open that file. Select your board and port as usual. Build and upload to your board. You will need to place your M4 board into bootloader mode by double pressing reset in order for the uploader to see it.

Connect via the Arduino serial terminal, screen or whatever way you use to connect to the Serial I/O on your board. When you do you'll be greeted by the REPL. You'll be greeted by a prompt of a number that is the number of lisp objects available (don't worry about it.. if it gets small you're running ourt of memory) followed by `>` and then wait for you to type code. Try `(+ 1 2)`. As soon as you type the closing `)` ulisp will evaluate what you typed and you should see `3.`

```
uLisp 2.3
23551> (+ 1 2)
3

23551>
```

Syntax

Lisp uses a somewhat different style of syntax. There are two kinds of data objects (that's a simplification, but it's sufficient for this guide) in Lisp: atomic objects (e.g. numbers or strings) and lists of data. A list is written as a sequence of data objects surrounded by a pair of parentheses. For example, a list of the numbers 0 through 9 is: `(0 1 2 3 4 5 6 7 8 9)`. Notice that there are no commas between data objects. As implied by the definition, lists can be recursive (since lists *contain* data objects and lists *are* data objects). E.g. `(1 "hi" (4 5 (6 7)))`. This is a list of three items. The first is a number, the second is a string, and the third is a list of two numbers and a list (which contains two numbers). Clearly lists don't have to contain a single kind of data object (i.e. they are heterogeneous).

Symbols are another kind of atomic data object. Lisp doesn't have variables in the same way many other languages do. Lisp has names (i.e. symbols) that can be bound to values (i.e. have a value connected). If we have the name `x` bound to the number `5`, whenever we refer to `x`, Lisp will look it up in it's list of bindings and find the value `5`, which it will then use. There are some exceptions to this, but they are mainly to do with making bindings. Lisp has a freer ideal about what makes a proper symbol (i.e. name) than most languages (I.e. what are valid identifiers in those languages). For example, in Lisp `+` is a perfectly fine symbol, as is `even?` or `change!`.

Another important kind of data object are boolean values, denoted in code as `t` (for True) and `nil` (for False).

If we type a list into the REPL it will get evaluated and the result returned. How does lisp evaluate a list? Let's say we type `(+ 1 2)` into the REPL. The system looks for a function as the first item. The symbol `+` is bound by default to a function that adds it's arguments together and returns the sum. The evaluator will call that function using the values of the rest of the items in the list as arguments to it (they are recursively evaluated). The result of that function call will be the result of the evaluation. So evaluating `(+ 1 2)` results in `3`.

Yes, you read that right. Functions are values. Obviously they can be bound to symbols, but they can also be passed into other functions or returned by functions.

Hardware control functions

Ulisp has a quite capable set of features and is tuned for small environments, microcontrollers in particular. As such, it supports Arduino style I/O controls with a set of functions for using the hardware. For the LED blinking example, we'll only need two:

(pinmode *pin mode*) Sets the input/output mode of a pin, and returns `nil`. Mode determines the direction/pullup: `0` or `nil` is `INPUT`, `1` or `t` is `OUTPUT`, and `2` is `INPUT_PULLUP`.

(digitalwrite *pin state*) Sets the state of the specified output pin. State can be `nil` (LOW) or `t` (HIGH).

Blink

Using the above along with the `delay` function (identical to the `delay` that the Arduino framework provides) we can write the standard blink function:

```
(defun blink ()
  (pinmode 13 t)
  (loop
   (digitalwrite 13 t)
   (delay 1000)
   (digitalwrite 13 nil)
   (delay 1000)))
```

A couple language points:

First of all, notice how Lisp code is simply lists. In fact, Lisp code is the same as Lisp data: the same syntax and internal structure is used for both. That lends itself to some incredibly powerful meta-programming techniques involving treating code as data (and data as code) that we won't get into here.

`defun` is the function that defines new functions. It takes the symbol to bind to the new function (`blink` in this case), a list of parameters (none in this case so the list is empty), and the code that is the body of the function (i.e. what to do when the function is called). It creates a function object and binds it to the name.

The `loop` function simply repeats the enclosed code indefinitely. It's like `while True` in Python or `while (1)` in C.

In the REPL we can execute the `blink` function by typing a list with the name `blink`. It needs to be in a list to be a function call; on it's own it's just a symbol and the value bound to it will be fetched. There are no parameters because our `blink` function requires none.

```
(blink)
```

It does what you'd expect: starts by setting up the pin for output (we're using the onboard LED on pin 13 for simplicity). It then loops, setting the pin high, waiting a second, setting it low, waiting a second, and repeats forever. To stop it, type `~`. This is just like you'd expect to see it in C or Python. The problem is that it's not very Lispy.

One of the fundamental idioms of Lisp is recursion, so much so that something called tail recursion optimization is a fairly standard feature in most Lisps. So an introduction to Lisp isn't complete without a discussion of recursion.

Recursion

Simple recursion (we won't worry about mutual recursion here) is when a function calls itself. Consider the factorial function. The factorial of n is n * (n-1) * (n-2) * ... * 1. We could do this with a loop, but it's more natural and elegant to do it with recursion. Factorial has two cases: `factorial(1) = 1` , and `factorial(n) = n * factorial(n-1)` . This assumes we'll only pass positive integers to `factorial` . So:

```
factorial(5)
5 * factorial(4)
5 * 4 * factorial(3)
5 * 4 * 3 * factorial(2)
5 * 4 * 3 * 2 * factorial(1)
5 * 4 * 3 * 2 * 1
5 * 4 * 3 * 2
5 * 4 * 6
5 * 24
120
```

A recursive factorial function in Lisp would look like:

```
(defun factorial (n)
  (if (= n 1)
    1
    (* n (factorial (- n 1)))))
```

This function works exactly as described above.

That `(= n 1)` check is called the base case, it's what stops the recursion. Always think about that first: how/when/why will the recursion stop? If you don't have that in place it's like an infinite loop.

Recursive solutions can be wonderfully simple and elegant, but at a cost. Each function call takes a bit of space, and they add up. Try to recurse too deeply and you run out of memory. The advantage of using a loop is that it uses a fixed amount of memory (not considering the case where you allocate more memory each time through the loop). Here again we have a somewhat nasty tradeoff. This one has a way around it, though: tail recursion optimization.

Tail Recursion

Tail recursion is the case when the last thing done in a function is the recursive call. In the factorial example above, the last thing done is the multiplication. We generally need to rearrange things a bit to accomplish tail recursion, often by passing a partial solution into the function. Here's a tail recursive version of factorial:

```
(defun f (n acc)
  (if (= n 1)
    acc
    (f (- n 1) (* n acc))))

(defun factorial (n)
  (f n 1))
```

In fact, we can define the recursive function inside the wrapper. This makes it completely local to the wrapper function.

```
(defun factorial (n)
  (defun f (n acc)
    (if (= n 1)
      acc
      (f (- n 1) (* n acc))))

  (f n 1))
```

Doing a similar analysis as we did above:

```
(factorial 5)
(f 5 1)
(f 4 5)
(f 3 20)
(f 2 60)
(f 1 120)
120
```

Even just looking at the trace, we can get a hint of the increased efficiency. In the non-tail recursive example, you can see the multiplication operations being saved and evaluated on *the way back*. In the tail recursive version, those multiplications are done right away, with the result passed into the recursive call. By the time we hit the base case and end the recursion, we have the final result and just have to return it.

The advantage of writing in a tail recursive way is that it can be simply and automatically optimized to run as if it were a simple loop, not actually making those nested recursive calls. Not all recursive algorithms can be written tail-recursively, but most can.

Revisiting Blink

For fun, let's rewrite `blink` in a tail-recursive way.

```
(defun blink (state)
  (pinmode 13 t)
  (digitalwrite 13 state)
  (delay 1000)
  (blink (not state)))
```

We could add a function if we wanted to get rid of the parameter, as well as the repeated `pinmode` call:

```
(defun blink ()
  (defun b (state)
    (digitalwrite 13 state)
    (delay 1000)
    (b (not state)))
  (pinmode 13 t)
  (b t))
```

The recursive function simply sets the pin's state to whatever was passed in, then waits and calls itself with the other state. The recursive call is the last thing done in the function so it's tail recursive and can be optimized automatically to have the same performance characteristics as the `loop` based version.

Resources

* ulisp with MetroM4Express support (https://adafru.it/BLs)
* Lisp implementations and resources (https://adafru.it/BLt)

Going further

Ulisp has all the hardware support you would expect/require from a language meant for microcontroller programming: serial, I2C, SPI, digital and analog I/O. It also has built-in SD card support.

As a quick example here's the code to reset and read temperature and relative humidity from an I2C si7021 temperature/humidity sensor.

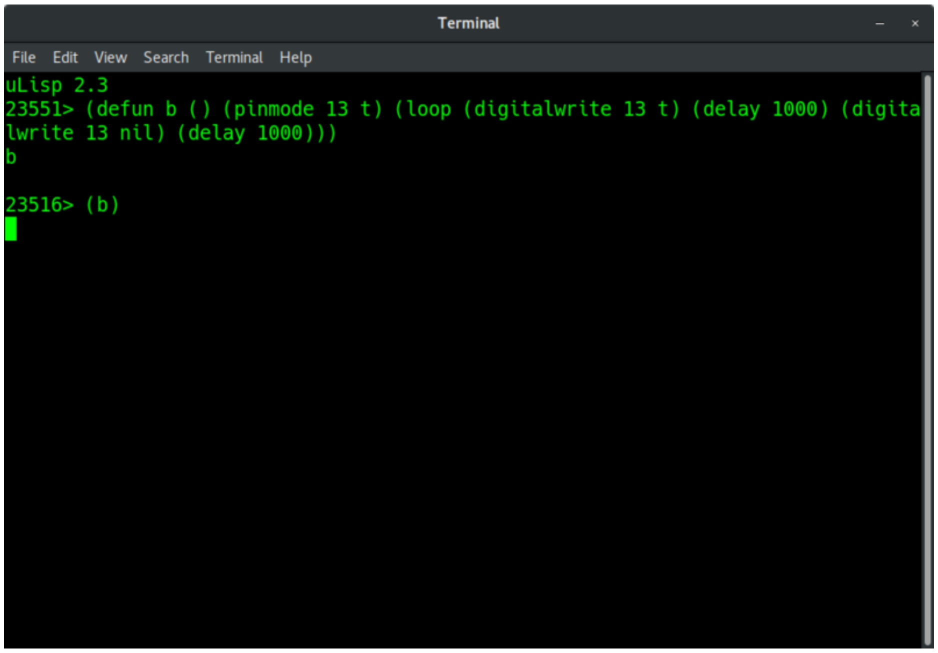
```
(defun si7021-reset ()
  (with-i2c (s #x40)
    (write-byte #xFE s))
  (delay 50)
  nil)


(defun si7021-temperature ()
  (with-i2c (s #x40)
    (write-byte #xF3 s))
  (delay 25)
  (with-i2c (s #x40 3)
    (let* ((hi (read-byte s))
           (lo (read-byte s))
           (ckecksum (read-byte s))
           (raw-temp (float (logior (ash hi 8) (logand lo #xFF)))))
      (- (/ (* raw-temp 175.72) 65536.0) 46.85))))


(defun si7021-humidity ()
  (with-i2c (s #x40)
    (write-byte #xF5 s))
  (delay 25)
  (with-i2c (s #x40 3)
    (let* ((hi (read-byte s))
           (lo (read-byte s))
           (ckecksum (read-byte s))
           (raw-temp (float (logior (ash hi 8) (logand lo #xFF)))))
      (- (/ (* raw-temp 125.0) 65536.0) 6.0))))
```

```
> (si7021-reset)
nil

> (si7021-temperature)
27.7967

> (si7021-humidity)
40.7339
```

If you've done any work writing I2C drivers in C or CircuitPython this should pretty much make sense. One neat feature about ulisp's approach is that it wraps the I2C connection up as a stream and can then use it's standard stream I/O functions to interact with it. The same is true with SPI, serial, and other stream-like interfaces.

In this example, the `with-i2c` function creates a stream to the I2C device at address 0x40 (64 decimal) and binds that to the symbol `s`. Within the scope of the `with-i2c` `s` is just a stream that can be read from and written to (depending on whether it was created to read/write: if a number is supplied after the I2C address, the stream is readable for that number of bytes). See the ulisp docs (https://adafru.it/BLu) for more detail.

Since ulisp is implemented as an Arduino sketch, it leverages all the portability capabilities of the platform: build it for the board you're using and all the interfaces are ready to use.



```
uLisp 2.3
23551> (defun b () (pinmode 13 t) (loop (digitalwrite 13 t) (delay 1000) (digita
lwrite 13 nil) (delay 1000)))
b

23516> (b)
```

Shortcomings

Ulisp is a solid, very portable implementation of a lisp for microcontrollers. Two shortcomings are that the interface of the REPL is simple and rather unforgiving. The other is that it's not the easiest to get code into the system. It would benefit from a better REPL interface (line editing, history, and such) as well as CircuitPython style SPI flash drive support for loading code onto it.