COMMON LISP
FAST, FUN, PRACTTICAL QUICKSTART
COMMON LISP
THE TUORIAL
THE TUTRRIAL

Common Lisp - The Tutorial

Fast, Fun, Practical Quickstart (With CLOG)

[See more at https://github.com/rabbibotton/clog/blob/main/LEARN.md](https://github.com/rabbibotton/clog/blob/main/LEARN.md)

The Journey - Part 1

Introduction

"If CL is ugly, why do I use it and write about it? Because Lisp is so powerful that even an ugly Lisp is preferable to using some other language." - Apr 30, 2002 - C.L.L - Paul Graham

Today we are going to embark on a journey together. I, a person with poor skills at writing[1] and the author of CLOG, the most Awesome GUI and Web Framework on the planet as of 2022[2], and you, an inquisitive individual looking to learn Common Lisp, CLOG and/or kabbalistic software design[3]. The goal of this journey is to quickly get you up to speed with enough Common Lisp to use this grotesquely beautiful language[4] with millions of parentheses[5] to write awesome software (with CLOG I hope), faster and more powerful than with any other language[6].

Rules of the Journey

1. **Ignore the parentheses and see only the indentations.**

The heart of Lisp is the S-expressions (aka the sexp[7]). A parenthesis followed by an operator followed by arguments and closed with another parenthesis. So (+ 1 2) results in a 3 and in almost every other language is 1 + 2. Data is also expressed in the same way e.g. (list 1 2 3)

[1] If only word processors had compilers, debuggers and profilers
[2] I know this to be true since CLOG is more awesome than GNOGA my Ada version from 2013 and until CLOG came around nothing was as awesome as it. I can also say this since there are only a few frameworks I know of that are designed from scratch for both web and gui use. Oh, one last proof, this fact has a footnote and is now on the internet so must be true.
[3] Which doesn't exist but if it did it would be in a Hebrew version of Lisp.
[4] Beauty is in the eye of the beholder, but Jean Ichbiah was an artist. To elaborate further, the author of Lisp was John McCarthy, a Litvach (a Lithuanian Ashkenazi Jew) and Jean Ichbiah was the author of Ada (a Turkish Sefardic Jew). In the world of jews the litvachs are stereotyped as the scientific types and the sefardic the artistic type, so you see this is all a very scientific analysis. I am a mixed breed so I write in Ada and Lisp.
[5] In 1987, I wanted to teach myself about AI programming. I was 11 then but I had already written very cool software in Assembly, Basic, Pascal, and lots of other languages but wanted more. So I asked Jack, the computer teacher at Nova University (they still have a k-12 division) who got me started programming with a bribe for the formula for how to draw circles if I could prove I would know what to do with it, what the best language would be, he said "prolog or Lisp?" I took one look at the parentheses and grabbed the prolog floppy.
[6] "Lisp is no harder to understand than other languages. So if you have never learned to program, and you want to start, start with Lisp." RMS
[7] Lisp really does have sexapeal.

the list containing four elements list 1 2 and 3, in memory it is stored as 1 2 and 3. The expressions nest (list 1 2 (list 2 4 (list 1 3) 4) 3) etc. This means lots and lots of parenthesis.

However, the human brain can parse them with the help of white space[8]:

```
(tagbody
  10 (print "Hello")
  20 (go 10))
```

In fact once you start looking at the indentation and forgetting about the parentheses Lisp starts to look like most languages.

```
(defun factorial (x)
  (if (zerop x)
    1
    (* x (factorial (- x 1)))))
```

This is an actual C program from the 1st International Obfuscated C Code Contest (1984)

```
a[900];     b;c;d=1     ;e=1;f;     g;h;O;      main(k,
l)char*     *l;{g=      atoi(*      ++l);       for(k=
0;k*k<      g;b=k       ++>>1)      ;for(h=     0;h*h<=
g;++h);     --h;c=(     (h+=g>h     *(h+1))     -1)>>1;
while(d     <=g){       ++O;for     (f=0;f<     O&&d<=g
;++f)a[     b<<5|c]     =d++,b+=    e;for(      f=0;f<O
&&d<=g;     ++f)a[b     <<5|c]=     d++,c+=     e;e= -e
;}for(c     =0;c<h;     ++c){       for(b=0     ;b<k;++
b){if(b     <k/2)a[     b<<5|c]     ^=a[(k      -(b+1))
<<5|c]^=    a[b<<5      |c]^=a[     (k-(b+1     ))<<5|c]
;printf(    a[b<<5|c    ]?"%-4d"    :"    "     ,a[b<<5
|c]);}      putchar(    '\n');}}    /*Mike      Laman*/
```

The simple regular syntax of Lisp and its use of parentheses is part of what makes Lisp so powerful, it is called homoiconicity. For now take my word on this, but it is omnipotent power made touchable by parentheses.

There is no place for discrimination! Do not judge a language based on its parentheses!

[8] Yes Lisp is from the 1960's and the classics like - 10 print "hello" 20 goto 10 was super cool and Lisp was served on punch cards too. Lisp, a little known fact, introduced throughout the history of software engineering most cool™ concepts we take for granted like if-then-else (as cond), automatic garbage collection, and was used for the first implementation of JavaScript (which is very Lispy).

2. Clear your mind, use the force luke! (Star Wars)

Lisp is a multi paradigm language. If it is a buzzword, Lisp invented it, has it (and did it better), or a few lines of code and will have it (it is the programmable language by design)[9]. It is not a mistake to write code using any paradigm as long as it is crisp and clean and no caffeine[10], i.e. readable and fitting the domain[11], error free and gets the job done[12].

- Functional - You bet, Lisp and Alonzo Church gave meaning to the letter lambda[13].
- Object Oriented - Your orientation is accepted here, we are all CLOS
- Procedural - It is our dirty little secret
- Structural - That goto example will haunt me
- Etc etc

3. Stability Matters

Lisp was specified in 1958, its first full compiler in 1962, and evolved into Common Lisp which became ANSI standard in 1994 and has been stable ever since[14]. The real secret of success[15] is building a foundation, perfecting it, only when absolutely necessary rewriting it[16].

The Ada version of CLOG has been around since 2013, CLOG the Common Lisp version is incrementally the same design with much more built on top. Some libraries CLOG sits on are older than most of you.

When you see a github Common Lisp project with a very old date, that just means it is stable and deserves a look.

Experience is fine wine, stability is the tortoise that wins over the hare, always[17].

[9] That is the case in Common Lisp and most Lisps, but not all Lisps are the same. From here on in, when we say Lisp we mean the ugly duckling that made it through years of college in the AI error, standardized as Common Lisp, then 30+ years of real world industrial experi.ence still in production and, because of experimentation with psychedelics, is not all baby skin, but looks awesome even though is the second oldest language in production use after Fortran.

[10] 7up, the source of that catchy phrase, is good stuff and not sticky like Sprite. Why are fat people discriminated against and gas stations and best buys only have diet Coke! Only skinny people drink diet cola, we larger folk want flavor, Diet Dr Pepper, Diet Fanta, etc. that is why we are fat!

[11] Define abstractions relevant to the problem domain and keep re-using them!

[12] "I am not a devotee of functional programming. I see nothing bad about side effects and I do not make efforts to avoid them unless there is a practical reason. There is code that is natural to write in a functional way, and code that is more natural with side effects, and I do not campaign about the question." RMS

[13] Some refer to CL as being only functional, yet on some wiki's they run out of disk space with those saying it is not a functional language. Just remember rule #2 clear your mind.

[14] Common Lisp and Dr. Who have much in common.

[15] "The Millionaire Mind" By Thomas J. Stanley inspired this line. Millionaires fix they don't throw.

[16] This of course is impossible when the language and libraries change like dirty diapers every year.

[17] Jonathan the Seychelles giant tortoise is 190+ years old. 'Nough said.

4. Community Matters

I have found in joining the Lisp community fantastic people with tons of experience all willing to contribute their experience and knowledge. They are brutal for their lingo, and rightfully so, beside the only way to communicate succinctly and make sure all on the same page, how we react to rebuke quickly shows who and what we are and if worth spending the time to communicate with.

I frequent #commonLisp on libera.chat thankfully they keep the channel strongly on topic and away from the greatest wastes of time on earth. Governmental politics. I am dbotton there and I have very strong feelings about never being anonymous online and that keeps me from saying things I should regret later[18].

There is also Lisp discord and https://www.cliki.net/ a wiki that will get you to many more resources.

The Open Source movement and Lisp are connected at the hip[19]. My fellow journeyer I have written insane amounts of free as in freedom lines of code and there is nothing more satisfying than knowing my code contributes to the advancement of us all[20]. The return with many years of doing this is making a living and loving life (the two rarely go together). Vertical development[21] is where there is always money to be made, but horizontal development needs to be free as in freedom, and sometimes, but not always, that means doing work for free as in beer.

5. Tools Matter

The success of a language is all about the tools and their open source status. Common Lisp is the most successful of all time with a crazy number of open source compilers (most still maintained!) and commercial compilers each with amazing tooling. Respect your tools. Assuming your code is the cause of the error not the compiler and your compiler will always be your friend.

[18] If ever in the Fort Lauderdale area say Hi.
[19] https://www.gnu.org/gnu/rms-Lisp.en.html
[20] Sadly the GPL can be abused by corporations to do great harm, entire languages and communities have been paralized by using the GPL to virus developer's software using their tools to prevent commercial use which harms all developers. I no longer use the GPL (BSD/MIT now) because of this and still very much respect the idea of keeping the tools free. On the ground though that is not what is being done, GPL was and often is used to poison our tools to prevent Vertical development (see next note).
[21] "Vertical" development is for example using CLOG to create a customer website or niche app and "Horizontal" development is general libraries and tools.

Where our journey will take us

As I am trench programmer[22], I am not the one to write the standard, the manual, the foundation for your development as a programmer, but I can excite you and get you writing Common Lisp with CLOG and that is my focus here.

I hope to cover:

- A minimal set of operators to write business/IT applications.
- Learn basic Lisp idioms
- Get you motivated to look into the depths of Lisp and Software Development in general.
- From early on learn about and how to use parallel computing - yes that is a minimum today!
- And of course learn to be a CLOGer[23]

[22] I could have a list of self deprecating things about myself and all would be true, but part of my goal with CLOG is attracting people to Common Lisp and to do that with this generation of whippersnappers it means whiz bang graphicy stuff, fresh and new stuff too focus them long enough to see it is worth it to work hard and Lisp will **payoff**. How You ask? With CLOG. You can already write "Horizontal Apps" faster and in more creative ways than ever before. That means you can turn apps into cash on an individual and corporate level and that means more developers in Common Lisp and that means more of another generation of people to advance the human race.

[23] Almost every culture has clogs of one type or another and where there are clogs there are cloggers dancing! A CLOGer loves to program and do cool stuff, for me writing software is my artistic outlet and I want to share my art and inspire others to use it and make their own.

The Symbol - Part 2

A hero ventures forth from the world of common day into a region of supernatural wonder: fabulous forces are there encountered and a decisive victory is won: the hero comes back from this mysterious adventure with the power to bestow boons on his fellow man.
- The Hero with a Thousand Faces, John Campbell

Introduction

I am glad I have not scared you off, we are going to train hard and fast and get you on your feet ready to make awesome happen. I am making some assumptions[24] that you have some experience with computers and programming ideas[25] and have emacs, slime, and sbcl installed[26]. There are many editors for Common Lisp but for this journey we are going to use emacs and slime, it is like learning to box by picking up chickens (Rocky II[27]).

Toto, I've a feeling we're not in Kansas anymore

Common Lisp is not like most languages you may have used[28], edit -> run compiler -> run executable, start again. Instead your editor (emacs in our case) is plugged in via slime (sort of like an umbilical cord) to a living breathing Lisp image containing your tools (sbcl the compiler, CLOG Builder, etc) and your code as you grow it. It is an organic process. When it comes time to deliver an executable (your baby) you **`save-lisp-and-die`**[29].

This system of development is far faster and certainly a lot more fun. You get to see results immediately and experiment on the spot. You can enter your code in one of two places to inject the imageith, either in the REPL or in text files that you read into the REPL.

The REPL - read–eval–print loop - is in many ways similar to an operating system's shell, so much so that there are Lisp alternatives for the shell[30]. In Lisp it is possible to write code that directly affects how code is read, how it is evaluated, and how the results are returned to you. All of that is beyond our journey.

[24] I know to ass-u-me is a bad idea, but I really love Common Lisp and think you will too.
[25] https://www.cs.cmu.edu/~dst/LispBook/book.pdf is a free book starting at the ground level and https://gigamonkeys.com/book/ is a good intro that is more practical. Another good source of sources https://stevelosh.com/blog/2018/08/a-road-to-common-lisp/
[26] If on windows grab https://portacle.github.io/ otherwise https://lisp-lang.org/learn/getting-started/
[27] https://www.youtube.com/watch?v=q7cDQY9wVF8
[28] Lisp was discovered not invented - http://www.paulgraham.com/rootsoflisp.html
[29] That really is how it is done… (sb-ext:save-lisp-and-die "hello.exe" :toplevel #'main :executable t)
[30] https://github.com/SquircleSpace/shcl

Let's start our Lisp image and start talking to it. Using emacs **`M-x slime`** starts slime and in most cases starts the new image. Once ready the REPL prompt is returned:

```
; SLIME 2.26.1
CL-USER>
```

For most of us the Lisp image resides on our laptop or the like, but it could be anywhere including a lunar lander on mars[31].

Type a simple command:

```
CL-USER> (print "hello")

"hello"
"hello"
```

The REPL returns the results of the function (print "hello") which is "hello" and the side effect (it is called a side effect because it doesn't affect our Lisp image but our world) of running the print function which is "hello" output to the `*standard-output*` stream which in this case is directed to our slime interface attached to our Lisp image.

Unless poking and prodding at our code already in the Lisp image most of the time we want to keep our code around in files.

In emacs execute **`C-x C-f`** and type a file name say like hello.lisp. The .lisp extension is most often used. Now we can enter our program in to our file:

```
(print "hello")
```

Save the file **`C-x C-s`**

In the REPL we can now "R"ead the contents of the file, which then will be "E"valuated and "P"rint the results.

```
CL-USER> (load "~/common-lisp/hello.lisp")

"hello"
T
```

Our side effect "hello" is output and T (the Lisp symbol for true, although any value that is not nil is considered true) is returned for the load operator.

[31] https://flownet.com/gat/jpl-lisp.html - Having a read-eval-print loop running on the spacecraft proved invaluable

Unlike other languages, files do not provide structure in a program[32]. Whatever you put in a file just gets pumped into the Lisp image when you load it as if you were typing it directly into the REPL.

The structure of a Lisp program is organized by "packages" of definitions for symbols. Symbols are associated in the reader with functions, macros, data, and more and give them a human readable name. There is only one Lisp image and everything lives in the same place, the symbols just refer to where things are in that image and packages group symbols together.

The REPL tells us which package is the default package of symbols we are using.

```
CL-USER>
```

In this case it is "CL-USER", which is just a name, but any standard implementation will already have that package defined.

The symbol name for print (with its home package) is - **`common-lisp:print`**. The cl-user package "**`uses`**" the common-lisp package of symbols and since that is our default package we don't have to write the package name part of the symbol name (**`common-lisp:`**) just the symbol **`print`**. Later we will discuss how to define our own packages[33].

I know my fellow journeyer this is all strange and different, a language that is alive, a pool of primordial ooze with sections of the goop identified by packages of symbols. It is important though to understand the nature of Lisp from now to not fall into the trap of thinking it is like other languages.

Symbols

Let's create some symbols and talk about them. You can create them in the REPL directly or store them in a file and load the file again in the future. There is an important set of keys to remember M-C-x. Using that emacs command allows you to just send the code you are working on directly to the Lisp image instead of rereading the entire file. For example in our hello.lisp file while near the "print" code hit M-C-x and "hello" will be output in our slime-repl window. The code was read and evaluated and the side effect was displayed from the `*standard-output*` stream.

[32] In ASDF systems files are part of their structure and will talk about them when we are ready to start building software.
[33] The package name is itself a symbol and *package* is a symbol that is associated with the symbol of the current default package the reader is using at the moment.

The first symbol we will create "main" will be associated with a function that we are familiar with already:

```
(defun main ()
  (print "hello"))
```

Now let's create a symbol *cool* that names a global dynamic variable with the value 123.

```
(defvar *cool* 123)
```

If I want to change the variable's value we use setf, Lisps general assignment operator.

```
(setf *cool* nil)
```

Nil is the null value and/or false. The symbol t is for true (more accurately anything not nil is true).

We can even change the value to our function from before.

```
(setf *cool* #'main)
```

Main is just a label pasted on a ball of goop in the list image. That the ball of goop is like any other data. #' is a "reader macro" that returns the function that a symbol names.

Symbols in Lisp besides being used to name balls of goop in a list image are a "type" in Lisp. When we use symbols for themselves, i.e. as a type we place an apostrophe before the symbol name:

```
'a-symbol
```

So we can say

```
(setf *cool* 'a-symbol)
```

You as the programmer decide what 'a-symbol means. It is not a string of characters, it is not a number, it is a symbol. Maybe one day you write a function when passed as an argument of the symbol 'love it paints flowers and when passed 'hate it paints skulls, passing "love" or "hate" would be an error as they are just talk, i.e. strings, not real 'love or real 'hate.

Symbols are case sensitive but the "reader" part of the repl turns all symbols to uppercase before evaluating them. So typing in to the REPL:

```
(equal 'a-symbol 'A-symbol)
```

Will return T.

You can though still create cased symbols by placing the symbol between pipes:

```
(equal '|a-symbol| '|A-symbol|)
```

Will return NIL.

```
(equal 'a-symbol '|A-SYMBOL|)
```

Will return T.

Technical Terms

The standard unit of interaction with a Common Lisp implementation is the ***form.***

Meaningful forms may be divided into three categories: **self-evaluating forms**, such as numbers; **symbols**, which stand for variables; and **lists**. The lists in turn may be divided into three categories: **special forms, macro calls, and function calls[34].**

Conclusion

I know my journey mate that we are going slow and theoretical so far, but seeing things the Lisp way is important for later on. Soon things will go much faster.

Summary

From Part 1

1. Lisp uses "LISt Processing"
2. The first element of the list is an operator and the remainder of the list are its arguments.

From Part 2

3. Lisp has a different development cycle than other languages and a different model of development, programs are grown.

[34] Common Lisp the Language, 2nd Edition Section 5.1

4. The top level of structure in Lisp programs is the package, which organizes symbols, not files.
5. A symbol is coded as **`package-name:symbol-name`** but if package-name is the same as the current default package you can refer to the symbol-name alone.
6. Symbols can name packages, functions, variables, and more.
7. Symbols can also be used as a data type as well by placing an apostrophe before the symbol name.

Functions - Part 3

You keep using that word. I do not think it means what you think it means.
- Inigo Montoya

Introduction

The speed of Lisp programming often is inconceivable. By growing things, testing in the REPL, playing with the results, incremental compiling of your code, things spring to life, but to be useful long term they need to be rooted in a package in our system, that is done by defining named functions. Functions in Lisp are also data and can be passed as arguments or returned from functions. Lisp is the father of Functional Programming.

Named Functions

Let's spin up our last system - hello-sys from the last part:

```
(ql:quickload :hello-sys)
```

Let's see our hello-package so far (we saved it in hello.lisp):

```
(defpackage :hello-package
  (:nicknames :hello-pkg)
  (:use :cl :clog)
  (:export :hello-world))

(in-package :hello-package)

(defun hello-private (body)
  "Create a div on new pages containing - hello world"
  (create-div body :content "hello world"))

(defun hello-world ()
  "Initialize CLOG and open a browser"
  (initialize 'hello-private)
  (open-browser))
```

There are two **named functions**. One, hello-world, is exported and the other not.
Named functions are defined with **`defun`**:

```
(defun my-func (parameter1 parameter2)
  "A DOC string used for documenting a named function"
  (+ parameter1 parameter2))
```

[+ is the addition operator, all the regular math symbols are there][35]

This function called my-func has two parameters. The doc string is optional, but my favorite part of Lisp, and I do my best to never write a function without one[36]. It adds the two parameters and the return value is the result of the last form of the function, i.e. their sum.

All the parameters and the specifiers (we will see &optional and &key in a moment) are called the functions **lambda list**.

A function can have no parameters as in hello-world as well and calling a function with no arguments[37] looks like this (hello-world) and with arguments each is separated by spaces and looks like this (my-func 1 2).

A function can have optional parameters:

```
(defun my-func2 (parameter1 &optional parameter2)
  "return the sum of parameters unless only parameter1 provided."
  (if parameter2
      (+ parameter1 parameter2)
      (parameter1)))
```

[The optional parameter2 if not set will be nil, i.e. false. The if operator is: `(if condition-form then-form &optional else-form)`]

So both `(my-func2 1)` and `(my-func2 1 2)` are valid.

A function can have named parameters:

```
(defun my-fun3 (parameter1 &key parameter2)
  "return the sum of parameters unless only parameter1 provided."
  (if parameter2
      (+ parameter1 parameter2)
      (parameter1)))
```

[35] We will be using [ ] after code snips or other code mentions to point out various operators we see in the snip its to learn Lisp as we go.
[36] I use mgl-pax https://github.com/melisgl/mgl-pax a great library that builds build beautiful documentation from doc strings and more. It helps in the one area that Lisp is poor at, interfaces/protocols between modules/packages. In the future I will write a tutorial to cover it.
[37] When you define a function you define it with parameters. When you call a function you call it with arguments.

[A named parameter not set will be nil]

A function with named parameters is called with `(my-fun3 1 :parameter2 2)` and can be left out `(my-func3 1).`

```
(defun my-fun4 (&key (say "Hello World"))
  "display the value of :SAY (Hello World default)"
  (print say) ; print say in machine readable format
  (princ say) ; print say in console readable format
  (terpri))   ; add new line (print and princ do not)
```

[Everything after a semicolon is a comment.]

The named parameter here `say` has a default value, so say will have the value when `(my-fun4)` is called of Hello World.

Anonymous Functions

Not all functions have names. They are anonymous functions created with a lambda expression:
  `(lambda (lambda-list) body)`

A lambda expression can be passed as an argument to a function just like data. Let's modify our hello-private function:

```
(defun hello-private (body)
  "Create a div on new pages containing - hello world
That when clicked changes color."
  (set-on-click (create-div body :content "hello world")
                (lambda (obj)
                  (setf (color obj) (rgb (random 255)
                                         (random 255)
                                         (random 255))))))
```

`set-on-click` from CLOG takes two arguments. The first is a clog object that is returned by `create-div` the second is an actual function, in this case our lambda expression, but could also have been a named function as well. When we pass a named function we prefaces it with #'symbol-name to say, pass as an argument the function named symbol-name.

[`setf`	- is the general assignment operator.
`color`	- is a object property of clog elements
`rgb`	- is a clog function that returns a value the color property understands.
`random`	- is random number (in our example 0 - 254)]

Conclusion

Functions transform data, they can have side effects like printing or graphics, and they can themselves be data. These properties of functions together are what make Lisp a Functional Programming Language[38].

Summary

From Part 1

1. Lisp uses "LISt Processing"
2. The first element of the list is an operator and the remainder of the list are its arguments.

From Part 2

3. Lisp has a different development cycle than other languages and a different model of development, programs are grown.
4. The top level of structure in Lisp programs is the package, which organizes symbols, not files.
5. A fully qualified symbol is coded as **`package-name:symbol-name`** but if package-name is the same as the current default package you can refer to the symbol-name alone.
6. Symbols can name functions, variables, and more.
7. Symbols can also be used as a data type as well by placing an apostrophe before the symbol name.

From Part 3

8. We navigate between packages using IN-PACKAGE
9. We define packages using DEFPACKAGE
10. A system is defined by a directory named the same as an .asd configuration file that describes what files should be loaded in a lisp image and what are the dependencies.

[38] Not to be confused with a subset, a pure functional language, that forces purely functional programming, ie. no side effects or mutable data allowed.

11. A system is loaded into the Lisp image with (ql:quickload :*system-name*)
    From Part 4

1. Functions can be named or anonymous (lambda expressions)
2. Functions can be treated as data
3. Lambda lists (the parameters) can be &optional or named &key

Packages and Systems - Part 4

As we live a life of ease (a life of ease)
Every one of us (every one of us)
Has all we need (has all we need)
Sky of blue (sky of blue)
And sea of green (sea of green)
In our yellow (in our yellow)
Submarine (submarine, aha)
– The Beatles

Introduction

Common Lisp evolved by cross breeding dialects of Lisp that were being used to write production real world software[39]. That makes Common Lisp a 30+ year stable hybrid with no need nor want of anyone to update to its 1984 standard and a powerful language to fight "fake tech" programmed with buzz words like web 3.0. "Real" real world software requires the ability to create modular code and to do so you need support in the language, this is achieved with packages in Common Lisp, and outside of the language, this is achieved with ASDF/QuickLisp systems.

Packages

As previously mentioned, the structure of a Lisp program is organized by "packages" of definitions for symbols. With those words we are ready now to begin growing our software and our **practical** knowledge of Lisp at the same time[40].

1. Open emacs
2. `M-x slime` to run slime
3. At the slime prompt `CL-USER>` we are going to type the command:
   ```
   (ql:quickload :clog)
   ```
   This will load CLOG into our lisp image and all the needed dependencies. We will learn more about quicklisp and its quickload in a bit. We will be using CLOG because it is no

[39] "Nevertheless this process has eventually produced both an industrial strength programming language, messy but powerful, and a technically pure dialect, small but powerful, that is suitable for use by programming-language theoreticians." https://dreamsongs.com/Files/Hopl2.pdf - The Evolution of Lisp
[40] As these are tutorials, they are to practically get you coding and playing with Common Lisp and CLOG. It is an unbelievably rich language with many brilliant ideas. Invest the time into learning Lisp properly with one of the many free books and resources available.

fun in 2022 to make believe we are using black and white CRTs[41] when we can use a "**C**ommon **L**isp **O**mificient **G**ui" from the start.

Navigating your Lisp Image

At this point our Lisp Image is alive and kicking with many cool packages (we are going to start exploring today). To change the default package we use IN-PACKAGE (by convention we copy the default behavior of the reader from our REPL and capitalize symbols when referring to them).

E.g. To change from CL-USER to CLOG-USER we use:

```
(in-package "CLOG-USER")
```

And now our default package is as indicated by our REPL:

```
CLOG-USER>
```

By convention we generally define our packages in all upper case (we will discuss defining packages soon). There is a practical reason for this which we shall learn soon.

There is a special package called "KEYWORD". Any symbol starting with a colon ':' is treated as a symbol from the "KEYWORD" package and is local to all packages. Like all symbols by default the reader upcases them.

```
CLOG-USER> :a_symbol
:A_SYMBOL
```

You can use a keyword symbol with IN-PACKAGE instead of a string and it will be turned into a string.

```
(in-package :clog-user)
```

Another alternative that can be used in IN-PACKAGE that you will see frequently is an uninterned symbol, i.e. a symbol that has no home package and is written #:package so the following is also valid:

[41] I was 8 when my father got me my first computer, a TRS-80 model 4. It had just come on the market and I was so excited, 64k memory!! The was double the ram of the computers in the lab at school. My father got me the deluxe model that came with 2 dual density 5 ¼" floppys 360K each! I spent every waking hour not at school glued to that machine and listening to my father scream how he could have gotten a Cadillac for the same price. My eyes were so messed from the CRT, between dry eyes, twitches, etc. Thankfully it seems to have left no long term effects on me… at least in the eyes.

```
(in-package #:clog-user)
```

I tend to use keywords, but to each their own[42]. Your Lisp image is your little creation.

It is a good practice though to fully qualify IN-PACKAGE with it's home package CL. Let see why:

```
CLOG-USER> (in-package :keyword)
#<COMMON-LISP:PACKAGE "KEYWORD">
KEYWORD> (in-package :clog-user)
; Evaluation aborted on #<UNDEFINED-FUNCTION IN-PACKAGE
{1005227763}>.
```

The package KEYWORD does not use the CL package (which is a nickname for the package COMMON-LISP) that contains the symbols with the Common Lisp language. The way out then is:

```
KEYWORD> (cl:in-package :clog-user)
#<PACKAGE "CLOG-USER">
CLOG-USER>
```

Creating Packages

Now that we can navigate our Lisp image, let's create our first package.

```
CL-USER> (defpackage :hello-package  ; create the package
           (:nicknames :hello-pkg)   ; alternate name
           (:use :cl :clog)          ; other packages to make local
           (:export :hello-world))   ; symbols exposed to the world
#<PACKAGE "HELLO-PACKAGE">
CL-USER> (in-package :hello-package) ; make our package the current
#<PACKAGE "HELLO-PACKAGE">           ; default
HELLO-PKG>
```

The comments give us a play-by-play.

Export is our "interface" or "protocol" to our package, ie what symbols will be available publicly in the lisp image and so accessible using the fully qualified symbol hello-package:hello-world or

[42] Using symbols to represent package names (which are strings in reality) keeps your code from "SCREAMING" at you and keywords are not affected by the current package. Using uninterned symbols works as well for this but I never got in the habit and see no need to.

in any DEFPACKAGE including hello-package in its :use without needing to specify its home package, i.e. hello-package and can use just hello-world.

Keep in mind that an export, exports the symbol. Symbols can mean many things, whatever it is now, it is publicly accessible. In Lisp, there is no strict enforcement of public and private. It is possible to access any symbol in a package by using two colons instead of one. So hello-package::hello-private would give access to the non-exported hello-private symbol.

Let's define two functions for the symbols we have talked about so far.

```
HELLO-PKG> (defun hello-private (body)
             "Create a div on new pages containing - hello world"
             (create-div body :content "hello world"))
HELLO-PRIVATE
HELLO-PKG> (defun hello-world ()
             "Initialize CLOG and open a browser"
             (initialize 'hello-private)
             (open-browser))
HELLO-WORLD
```

Even though we haven't yet explained the details of these functions, it is clear that violating the protocol and calling the function hello-private would be a mistake, as hello-world is where the initialization is taking place. So with great power comes great responsibility.

ASDF Systems and QuickLisp

Packages allow us to develop the internals of our "system", i.e. our Lisp image. ASDF and QuickLisp provide the means to make that system reproducible from outside the Lisp image.

ASDF allows us to define where our code is located in the real world and how to reconstruct the Lisp image from scratch. QuickLisp sits on top of ASDF and retrieves if needed over the internet any dependencies of your system and any dependencies of those dependencies. There are other options to ASDF and QuickLisp but for most needs they do an excellent job.

Let's turn the bit of code we created before into a full system.

1. We need to create a directory for our system in a location ASDF knows about. The directory ~/common-lisp is built in and what we will use. (If using portacle portacle\projects). The directory ~/.quicklisp/local-projects is also configured on many systems and you can add any directory by running in the Lisp image as well (push #P"path/to/dir/of/projects" ql:*local-project-directories*)
2. Create the directory hello-sys in ~/common-lisp

3. Next we need to copy our code we typed before into a file. C-x C-f and create the file ~/common-lisp/helllo-sys/hello.lisp and copy and paste to create:

```
(defpackage :hello-package
  (:nicknames :hello-pkg)
  (:use :cl :clog)
  (:export :hello-world))

(in-package :hello-package)

(defun hello-private (body)
  "Create a div on new pages containing - hello world"
  (create-div body :content "hello world"))

(defun hello-world ()
  "Initialize CLOG and open a browser"
  (initialize 'hello-private)
  (open-browser))
```

4. Next we need to create our .asd file that tells ASDF what files to load into a list system and what the dependencies are. It needs to have the same name as our directory which is the name of the system, so the file name is ~/common-lisp/hello-sys/hello-sys.asd and the contest is:

```
(asdf:defsystem #:hello-sys
  :description "Common Lisp - The Tutorial Part 3"

  :author "david@botton.com"
  :license  "BSD"
  :version "0.0.0"
  :serial t
  :depends-on (#:clog)
  :components ((:file "hello"))) ; <- notice no .lisp used
```

5. Let's let ASDF recalculate the available systems - (asdf:clear-source-registry) or we can reset our lisp image using M-x slime-restart-inferior-lisp
6. The we request QuickLisp to load our brand new system - (ql:quickload :hello-sys)

```
To load "hello-sys":
  Load 1 ASDF system:
    hello-sys
; Loading "hello-sys"
..................................................
[package hello-package]
```

```
(:HELLO-SYS)
```

7. We can now try out our complete system:

```
CL-USER> (hello-pkg:hello-world)
Hunchentoot server is started.
Listening on 0.0.0.0:8080.
HTTP listening on    : 0.0.0.0:8080
HTML Root            : /home/dbotton/common-lisp/clog/./static-files/
Boot js source       : compiled in
Boot file for path / : /boot.html
NIL
NIL
0
```

And a browser should open on most computers. If it does not go to http://127.0.0.1:8080/ and you will see the famous words Hello World.

Summary

From Part 1

1. Lisp uses “LISt Processing”
2. The first element of the list is an operator and the remainder of the list are its arguments.

From Part 2

3. Lisp has a different development cycle than other languages and a different model of development, programs are grown.
4. The top level of structure in Lisp programs is the package, which organizes symbols, not files.
5. A fully qualified symbol is coded as **`package-name:symbol-name`** but if package-name is the same as the current default package you can refer to the symbol-name alone.
6. Symbols can name functions, variables, and more.
7. Symbols can also be used as a data type as well by placing an apostrophe before the symbol name.

From Part 3

8. We navigate between packages using IN-PACKAGE
9. We define packages using DEFPACKAGE
10. A system is defined by a directory named the same as an .asd configuration file that describes what files should be loaded in a lisp image and what are the dependencies.

11. A system is loaded into the Lisp image with (ql:quickload :*system-name*)

Control Flow - Part 5

And if you listen very hard
The tune will come to you at last
When all is one and one is all, that's what it is
To be a rock and not to roll, oh yeah
And she's buying a stairway to heaven
– Led Zeppelin - Stairway to Heaven

Introduction

The scaffolding of a lisp program is the external asdf system and files (the cross bracing) and the internal packages (end frames) with their symbol exports (platforms) connecting the packages to each other. Once you have scaffolding, it's "Time to make the donuts"[43].

In Lisp you neither develop top down nor bottom up, it is organic, a donut. **Lisp loves donuts.** This tutorial started with the outer structure of a Lisp program so that you can experience that, not just testing things at the REPL, but being able to produce a Lisp image that can become a product or a service, adding features and trying things out as you go[44] with something you can deliver[45].

The Cycle of Life

So far we have seen that at the REPL we can type and call a named function and control flows to it. We also saw that control can flow from an event like clicking a mouse to a named or anonymous function as well. The parameters of a function control the flow into the function.

```
(defun give-it (here)
  (princ here)
  (print here))
```

The control flow in `give-it`:

[43] Dunkin' Donuts 1984 Fred the Baker - https://www.youtube.com/watch?v=IYRurPB4WA0 - watch to the end 'cause **Lisp loves donuts.**
[44] I love custard and jelly donuts so I can see why people want to eat the jelly first, to work on algorithms and unique cool macros, but reality is you only get a drop of jelly in the donuts. So make good dough first and then can get sticky with the profits. **Lisp loves donuts**.
[45] 'Cause thanks to covid-19 almost no one eats out anymore.

1. So at the REPL I can say `(give-it 1)` and a transfer of control from the REPL to the function `give-it` takes place, what flows is the argument, the number 1.
2. Once we enter the function the parameter `here` is **bound** to the argument flowing in, i.e. the number 1. (Keep in mind `here` can be bound in Lisp to any object of any type[46].)
3. The flow enters the body and we process one form at a time. In `give-it` this means the `princ` form followed by the `print` form.
4. The last form `give-it` evaluates to the value of `here` and that is what is returned by `give-it`
5. Control us returned to the REPL again. **Lisp loves donuts.**

That is the full cycle of life for `give-it` with every call.

Pebbles in the Stream

The flow within the body of a function (#3 above) is where the interesting things happen in a program and the focus for the next few tutorials.

Lisp has a crazy number of "block" forms, i.e. a sequence of forms executed one after the other like #3 above, most of which are rarely used (`prog, prog*, locally, tagbody, prog1, prog2, block`)[47] and we will discuss `progn` now and `let/let*` in the next tutorial.

A `progn` in Lisp is just a "block", and a "block" in Lisp is called an implicit progn. **Lisp loves donuts.**

Every form in lisp evaluates to a value, but often it is the side effects (like print text or painting graphics) we want and the return value is irrelevant. A `progn` discards every evaluation but the last[48].

What then is the purpose of having the operator `progn`, the most common use is to allow side effects to happen in the middle of the flow of control.

Let's open our hello.lisp from our hello-sys and modify our hello-private:

[46] This is power and danger, but beyond the scope of this tutorial. Someday I will follow up with a tutorial on static vs dynamic typing. Lisp is a strongly typed dynamic language and compilers like sbcl provide strong static typing. Lisp is all about flexibility and options, some programs only need quick implementation and are tolerant of errors, some need mission critical early bug detection and that turns a small program into a large one.
[47] Lisp is the programmable language and these represent distinct building blocks of higher level language constructs and used more often in macros.
[48] The name progn means return the n'th forms evaluation of the prog[ram] and prog1 means return the first forms evaluation and prog2 means return the second forms evaluation.

```
(defun hello-private (body)
  "Create a div on new pages containing - hello world
That when clicked changes color."
  (set-on-click (create-div body :content "hello world")
                (lambda (obj)
                  (setf (color obj) (progn
                                      (terpri)
                                      (princ "Setting color to:")
                                      (print (rgb (random 255)
                                                  (random 255)
                                                  (random 255))))))))
```

Let's spin up our hello-sys again:

```
(ql:quickload :hello-sys)
(hello-package:hello-world)
```

In this modification we inserted code with side effects (terminal printing) in the middle of an assignment to the `color` property of `obj`. As you can see this can be a nifty little trick to use.

The other more common use of `progn` is to turn the functionally inclined `if` operator[49] we already saw into a control flow operator:

```
(if this-item
  (progn
    (do-this-form)
    (and-this-form))
  (progn
    (else-this-one)
    (and-this-one)))
```

Go With the Flow

So `if` can branch the flow of your code in two directions, but `cond` allows many conditions and so many branches of flow and each direction has its own implicit progn.

Let's modify our hello-private again to become a little game:

[49] `if` in its pure form is about conditionalizing assignment. Lisp, like life, is all about choices, though some choices can make future choices harder, some life shorter, and some just terminate you on the spot. "Experience is the worst way to learn, reading is the intelligent way to learn, but listening to your elders is the wisest way of all". - Wise sayings of the wise guy, me.

```
(defun hello-private (body)
  "Create a div on new pages containing - hello world
That when clicked changes color."
  (set-on-click (create-div body :content "CLICK ME TO PLAY")
                (lambda (obj)
                  (setf (color obj) (cond ((equal (random 10) 1)
                                           (setf (text obj) "--(O)-(O)--")
                                           (print "RED LIGHT!")
                                           (rgb 255 0 0))
                                          (t
                                           (setf (text obj) "--(X)-(X)--")
                                           (print "I'm not looking..")
                                           (rgb 0
                                                (random 255)
                                                (random 255))))))))
```

You can use M-C-x and then refresh your browser if still open or open to http://127.0.0.1:8080

Here we have our `cond` operator that is inclined to control flow being used for its functional result, the last evaluation of each of the implicit `progn`s.

Let's analyze the `cond` :

```
(cond (something-evaluating-to-anyhting-but-nil
        (form-one)
        (form-two))
      (something-else-evaluating-to-anyhting-but-nil
        (form-one)
        (form-two))
      (t
        (form-one)
        (form-two)))
```

So any number of conditions can be used, and you can use `t`, a built-in symbol that is not nil for the catch all.

Lisp also has `when` and `unless` both of implicit progns

```
(when some-value-true
  (form-one)
  (form-two))

(unless some-value-false
  (form-one)
  (form-two))
```

Comparing "Equal"s

When comparing in lisp there are two main operators (that also work for numbers, strings and symbols): `equal` and `equalp`

`equal` - Is a case sensitive comparison ("hello" and "Hello" are **not** equal) and type sensitive comparison (10 and 10.0 are **not** equal).

`equalp` - Is a case insensitive comparison ("hello" and "Hello" **are** equalp) and type insensitive comparison (10 and 10.0 **are** equalp).

Other Equality Comparisons

These two additional equality operators that exist for **efficiency**, i.e. if you don't understand them just use `equal` for now.

`eq` - Two items are `eq` if they are the same object in memory (like a pointer compare). So the same symbols are `eq` and lists which are passed by reference are eq if two variables contain the same reference, e.g. `(setf x '(1 2 3)) (setf y x) (eq y x)` is true, i.e. x and y both point to the same list in memory. `eq`'s main use is to compare symbols `(setf x 'sym) (when (eq x 'sym) (print "yip"))` The result of comparing anything else is "undefined".

`eql` - Just like `eq` but can also compare numbers (10 and 10.0 are not eql by the way) and individual characters (case insensitive and can not be used for strings).

Summary

Simplified control flow is

external event -> function -> body -> progn -> if / cond / when / unless branching
=> return evaluation of last form in progn

Globals and Lists - Tutorial 6

Wax on, right hand. Wax off, left hand. Wax on, wax off.
Breathe… in through nose, out the mouth.
Wax on, wax off.
Don't forget to breathe. Very important.
Wax on, wax off. Wax on, wax off.
– Mr. Miyagi

Introduction

Lisp is named Lisp for "**Lis**t **P**rocessing" and we have seen the main input to the compiler are lists in a form, well, called the **form**, (operator argument1 argument2…). Like in Karate, forms, well practiced sets of moves, gives you power in a fight to know how to respond with well practiced attacks and defenses, the use of the **form** in Lisp gives you power to manipulate code as data and data as code[50], homoiconicity, a power other languages can only dream of, and you shall taste[51] of it in this tutorial.

The **EVIL** Globals

Before we go further, I am compelled to introduce you to three top-level-forms that sometimes are the source of EVIL and sometimes they LIVE for a greater purpose… "Dynamic Scope", a cool and bizarre twist of history that changes EVIL => LIVE, born of history and fairy magic, and for another tutorial, is one such good purpose. We shall reveal others now.

```
(defvar *symbol-for-global-variable* "optional initial value"
 "I LOVE DOC strings")
```

The convention for global variables and parameters is to place earmuffs around the variable name, ***my-var*.** This is to let everyone know "I have evil potential". Not everyone agrees with this convention but I suggested "Until you fully know both sides of an argument, don't form an opinion." (Wise Words from a Wise Guy - me).

What is the "evil potential" of a global variable? It is, that it is global. You can read and change the value of the variable from any function or even the REPL of the Lisp image at any time and even concurrently[52] and that may affect any use of that variable silently and with no notice.

[50] The spice is the worm, the worm is the spice.
[51] It tastes like chicken. (All code does)
[52] A tutorial on currency is for sure coming.

Globals have power as well. In CLOG's second demo (clog:run-demo 2) a chat app, a global hash[53], allows all the users to interact with each other in just a few lines of code.

Since Lisp is image based, not file -> compile -> run based, there is something unique about these global variables. If I tell the compiler to compile the defvar with a new "optional initial value" it won't change the value that already exists for the global variable, that option is only set the first time it is evaluated into our Lisp image. In others M-C-x has no real function on a defvar after the first time (the DOC string will change if modified[54] :).

```
(defparameter *symbol-for-global-parameter* "value"
 "I LOVE DOC strings")
```

A parameter is a global variable that every time you recompile the definition it is bound again to "value". In many ways this is the purpose of a constant in most languages, a value is not usually changed by the application. The nature of Lisp is that you may want to make changes live on a setting and so you can.

```
(defconstant symbol-for-global-constant "initial value"
 "I LOVE DOC strings")
```

A constant never changes and the compiler will raise an error if you try. In most cases beyond giving a name to a numerical number this is unlikely the best choice and should use defparameter.

List Power

Lists of data are entered with the `list` operator:

```
(list 1 2 3 4)
=> (1 2 3 4)
```

[=> means evaluates to (i.e. what would be in memory)]

The form returns a list object containing 4 elements that are numbers, Lists can contain any object:

```
(list 1 #\a "2" 'symbol-name #'print)
   =>(1 #\a "2" SYMBOL-NAME #<FUNCTION PRINT>)
```

[53] A hash is like a list where each element has a name, we will get there.
[54] In the future we will do a tutorial on introspection tools as the doc strings are accessible to your code too.

[The REPL can't print what a function looks like so it uses a #<> notation to represent it.]

This form returns a list with five elements, each a different type, a number, a character, a string, a symbol, and a function.

There are a HUGE number of operators for operating on Lists, some even have duplicate names for the same functionality. Yes, now you know it too, the ugly truth of Common Lisp… with age and wisdom comes skin wrinkles. Common Lisp is a mix breed of many many previous Lisps (some with names as ridiculous as **nil**) and it picked up along the way slang words (only some are four letters). No one is forcing you to use that slang, but it is not hard to figure out what someone says, you just use: http://l1sp.org the Lisp "the search engine" lookup or can ask the REPL with `(describe 'symbol-to-describe)`. Another good resource is https://jtra.cz/stuff/lisp/sclr/index.html that lists commonly used parts of Common Lisp.

We will use the REPL for now to demonstrate a few to get started. First let's set up a list in a global variable to work with:

```
(defvar *mylist* (list 1 2 3 4 5))
```

The first element:

```
(car *mylist*)
=> 1
```

or

```
(first *mylist*)
=> 1
```

Get the rest of the list after the first:

```
(cdr *mylist*)
=> (2 3 4 5)

(rest *mylist*)
=> (2 3 4 5)
```

Get the second, third, fourth, and fifth:

```
(second *mylist*)
=> 2
(third *mylist*)
=> 3
```

```
(fourth *mylist*)
=> 4
(fifth *mylist*)
=> 5
```

Get the n'th element:

```
(nth 3 *mylist*)
=> 4 ; Notice 4 - nth starts at 0
```

Append two lists:

```
(append *mylist* (list 1 2 3))
=> (1 2 3 4 5 1 2 3)
```

Reverse list:

```
(reverse *mylist*)
=> (5 4 3 2 1)
```

You will notice that *mylist* was not modified by any of the above operators, i.e. there are no side effects.

There are a few operators with side effects worth knowing now:

Push an element to start of list:

```
(push 99 *mylist*)
=> (99 1 2 3 4 5)

*mylist*
=> (99 1 2 3 4 5)
```

And as a side effect *mylist* was changed.

Pop an element off of list:

```
(pop *mylist*)
=> 99

*mylist*
=> (1 2 3 4 5)
```

And as a side effect *mylist* was changed.

A Shortcut

Using the list operator to create a list is the "long" way:

(list 1 2 'symbol-1 'symbol-2 "a string")

For convenience you can use quote to say what comes next, don't evaluate, it is data.

(quote (1 2 symbol-1 symbol-2 "a string"))

And further can simplify that with the single quote **'**

'(1 2 symbol-1 symbol-2 "a string")

Which evaluates to a list like the operator list but also removes the need to use a single quote to treat the symbol name as a symbol.

setf General Assignment and Lists

Lisp has an operator called setf that is more than a "this equals that". You can for example set individual elements:

```
(setf (second *mylist*) 99)
=> 99

*mylist*
=> (1 99 3 4 5)
```

Of course we can just replace the current value with:

```
(setf *mylist* '(1 2 3))
=> (1 2 3)
```

Closures, Loops and Strings - Part 7

You spin me right 'round, baby, right 'round
Like a record, baby, right 'round, 'round, 'round
You spin me right 'round, baby, right 'round
Like a record, baby, right 'round, 'round, 'round
– Dead or Alive

Introduction

Life tends to get boring when things are going in circles, but in the metaverse loops are action and functions are windows in.

Scope and Closures

Global symbols have a scope that allows any part of the lisp image to reach or modify their value. All other symbols have a lexical scope, i.e. they symbols are bound to values where they are textually defined and when that textual block of code is over, the symbols are no longer bound to those values. For example the parameter symbols are bound only within the function body to the arguments used to call the function until the function returns.

We define local symbols using let (or let* if there is a need for one definition to refer to another as part of their definition) which creates an implicit progn where they will be defined until completion of that progn. To illustrate:

```
(defun my-func (param1 param2)
  (let ((sum (+ param1 param2)))
    (print sum)))
```

[Syntax:

```
  (let ((symbol1 initial-value)
        (symbol2 initial-value)
        symbol2-noinit)
     body)                           ]
```

param1 and param2 are bound from the start of my-func until it's implicit progn is complete, i.e. the last right parenthesis is closed. sum is bound until let's implicit progn is complete.

Lisp also allows defining locally named functions in the same manner as let/let* called flet/labels (the reason for the "labels" name instead of flet* is historical, like car[55] and cdr[56]).

It is possible to return a function like other data and even though the source scope is lexically closed (the last right parenthesis closed) the environment where a function is defined lives on with it and why it is called a closure.

```
(defun my-adder (grow-by)
  (let ((sum 0))
    (lambda ()
      (incf sum grow-by))))

(defvar *two-counter* (my-adder 2))
(defvar *two-counter-two* (my-adder 2)) ; a second adder by 2
(defvar *three-counter* (my-adder 3))

(funcall *two-counter*)
=> 2

(funcall *two-counter*)
=> 4

(funcall *three-counter*)
=> 3

(funcall *two-counter-two*)
=> 2
```

The my-adder function is a "factory" for adders and with that you have the OO factory pattern and the fact that OO existed in Lisp from that 1960s in some form.

Loops

The other category of control flow operators after functions and branching is loops. Lisp has many flavors of loops and even an advanced version of loop with its own language that we will have a separate tutorial on[57].

loop..return

If return is never called loops forever.

[55] "**C**ontents of the **A**ddress part of **R**egister number" http://jmc.stanford.edu/articles/lisp/lisp.pdf
[56] "**C**ontents of the **D**ecrement part of **R**egister number"
[57] There are two extra languages in Lisp, format and loop and we will address both in next tutorial

```
(let ((n 0))
  (loop
    (princ ".")
    (if (> n 10)
        (return n)
        (incf n))))
```

dotimes

The loop repeats n times, n equals 0 .. n-1 on each loop.

```
(dotimes (n 10)
  (princ "."))
```

dolist

Each loop n is bound to the next element until all elements are done.

```
(dolist (n '(1 2 3 4 5))
  (princ n))
```

do

Do allows for multiple loops at once. Each loop is defined with:
```
 (a-symbol starting-val how-to-change-a-symbol)
```

```
(do ((x 1 (+ x 1))    ; loop 1
     (y 10 (- y 1))) ; loop 2
    ((> x 10))       ; terminate loops when true
  (princ x)
  (princ " - ")
  (princ y)
  (terpri))
```

String Processing

Strings are created using double quotes. Strings in Lisp are a single dimension array of characters.

Lisp provides string specific operators in addition to the more generic one used for sequences, arrays and equality.

Common String Operators:

```
(length x)                      ; string length
(string= x y)                   ; case sensitive equality

(string-upcase x)               ; return uppercase vs of x
(string-downcase x)             ; return lowercase vs of x
(string-capitalize x)           ; return x with each word capitalized
(string-trim x)                 ; trim white space from left and
right
(string-left-trim x)            ; trim white space from left
(string-right-trim x)           ; trim white space from right
(reverse x)                     ; reverse contents of string

(subseq x s e)                  ; return substring of x from s to e
(setf (subseq x s e) y)         ; replace in x fromm s to e with
                                ; contents of y that fit in s to e

(concatenate 'string x y)       ; concatenate strings

(string #\x)                    ; convert character to string
(character "x")                 ; convert string to character

(parse-integer "123")           ; convert and integer string to
number
(read-from-string "123.232"     )    ; reads data from a string -
warning
(write-to-string x)             ; convert evaluated x to a string
```

Characters are written in Lisp as #\z - z being the character that it designates.

```
(char my-string x)              ; return character at x
(setf (char my-string x) y)     ; replace char at x with y

(char= x y)                     ; case sensitive equality

(code-char x)                   ; return char for value x
(char-code x)                   ; return value for char x
```

Some Character Constants:

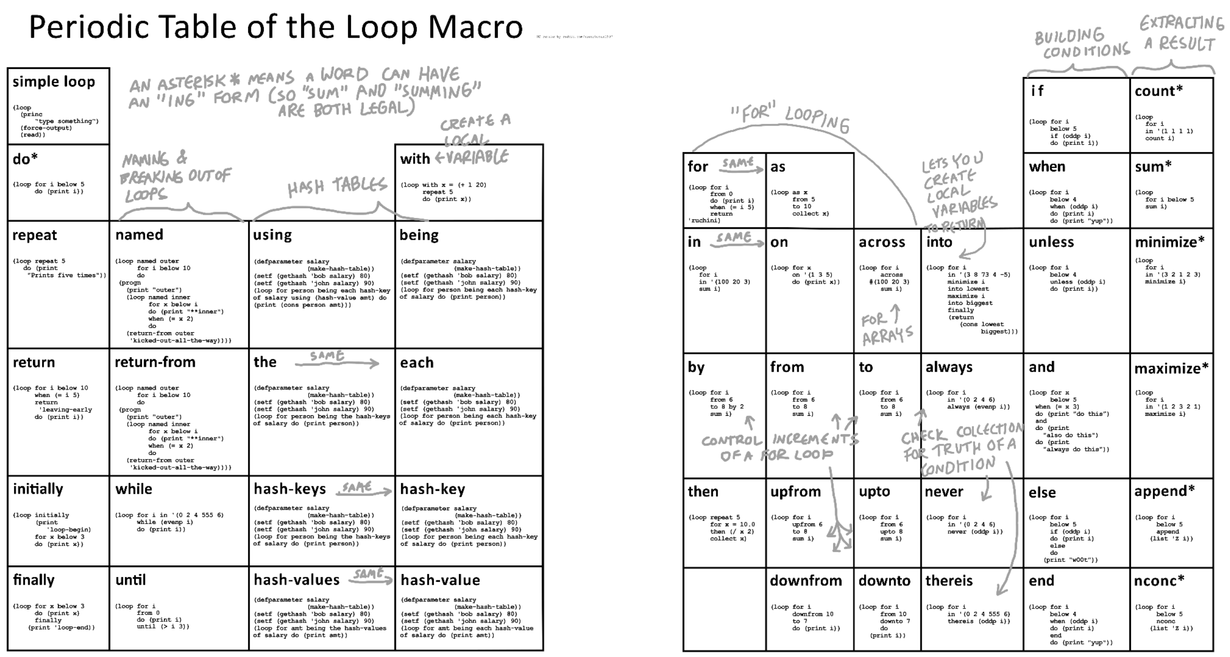
```
#\space
#\newline
#\linefeed
#\return
#\tab
#\backspace
#\rubout
```

Loop and Format - Part 8

Pretty woman, walkin' down the street
Pretty woman the kind I like to meet
Pretty woman I don't believe you, you're not the truth
No one could look as good as you, mercy
– Roy Orbison

Introduction

Only a mother could love this loop table:


Periodic Table of the Loop Macro
AN ASTERISK * MEANS A WORD CAN HAVE AN "ING" FORM (SO "SUM" AND "SUMMING" ARE BOTH LEGAL)
NAMING & BREAKING OUT OF LOOPS
HASH TABLES
CREATE A LOCAL VARIABLE
SAME
"FOR" LOOPING
LETS YOU CREATE LOCAL VARIABLES TO RETURN
FOR ARRAYS
CONTROL INCREMENTS OF A FOR LOOP
CHECK COLLECTION FOR TRUTH OF A CONDITION
BUILDING CONDITIONS
EXTRACTING A RESULT

simple loop
```
(loop
  (princ
     "type something")
  (force-output)
  (read))
```

do*
```
(loop for i below 5
      do (print i))
```

with
```
(loop with x = (+ 1 20)
      repeat 5
      do (print x))
```

repeat
```
(loop repeat 5
   do (print
    "Prints five times"))
```

named
```
(loop named outer
      for i below 10
      do
 (progn
   (print "outer")
   (loop named inner
         for x below i
         do (print "**inner")
         when (= x 2)
         do
   (return-from outer
    'kicked-out-all-the-way))))
```

using
```
(defparameter salary
                (make-hash-table))
(setf (gethash 'bob salary) 80)
(setf (gethash 'john salary) 90)
(loop for person being each hash-key
of salary using (hash-value amt) do
(print (cons person amt)))
```

being
```
(defparameter salary
                (make-hash-table))
(setf (gethash 'bob salary) 80)
(setf (gethash 'john salary) 90)
(loop for person being each hash-key
of salary do (print person))
```

return
```
(loop for i below 10
      when (= i 5)
      return
       'leaving-early
      do (print i))
```

return-from
```
(loop named outer
      for i below 10
      do
 (progn
   (print "outer")
   (loop named inner
         for x below i
         do (print "**inner")
         when (= x 2)
         do
   (return-from outer
    'kicked-out-all-the-way))))
```

the
```
(defparameter salary
                (make-hash-table))
(setf (gethash 'bob salary) 80)
(setf (gethash 'john salary) 90)
(loop for person being the hash-keys
of salary do (print person))
```

each
```
(defparameter salary
                (make-hash-table))
(setf (gethash 'bob salary) 80)
(setf (gethash 'john salary) 90)
(loop for person being each hash-key
of salary do (print person))
```

initially
```
(loop initially
      (print
         'loop-begin)
      for x below 3
      do (print x))
```

while
```
(loop for i in '(0 2 4 555 6)
      while (evenp i)
      do (print i))
```

hash-keys
```
(defparameter salary
                (make-hash-table))
(setf (gethash 'bob salary) 80)
(setf (gethash 'john salary) 90)
(loop for person being the hash-keys
of salary do (print person))
```

hash-key
```
(defparameter salary
                (make-hash-table))
(setf (gethash 'bob salary) 80)
(setf (gethash 'john salary) 90)
(loop for person being each hash-key
of salary do (print person))
```

finally
```
(loop for x below 3
      do (print x)
      finally
    (print 'loop-end))
```

until
```
(loop for i
      from 0
      do (print i)
      until (> i 3))
```

hash-values
```
(defparameter salary
                (make-hash-table))
(setf (gethash 'bob salary) 80)
(setf (gethash 'john salary) 90)
(loop for amt being the hash-values
of salary do (print amt))
```

hash-value
```
(defparameter salary
                (make-hash-table))
(setf (gethash 'bob salary) 80)
(setf (gethash 'john salary) 90)
(loop for amt being each hash-value
of salary do (print amt))
```

if
```
(loop for i
      below 5
      if (oddp i)
      do (print i))
```

count*
```
(loop
   for i
   in '(1 1 1 1)
   count i)
```

for
```
(loop for i
      from 0
      do (print i)
      when (= i 5)
      return
'ruchini)
```

as
```
(loop as x
      from 5
      to 10
      collect x)
```

when
```
(loop for i
      below 4
      when (oddp i)
      do (print i)
      do (print "yup"))
```

sum*
```
(loop
   for i below 5
   sum i)
```

in
```
(loop
   for i
   in '(100 20 3)
   sum i)
```

on
```
(loop for x
      on '(1 3 5)
      do (print x))
```

across
```
(loop for i
      across
   #(100 20 3)
      sum i)
```

into
```
(loop for i
      in '(3 8 73 4 -5)
      minimize i
      into lowest
      maximize i
      into biggest
      finally
      (return
         (cons lowest
               biggest)))
```

unless
```
(loop for i
      below 4
      unless (oddp i)
      do (print i))
```

minimize*
```
(loop
   for i
   in '(3 2 1 2 3)
   minimize i)
```

by
```
(loop for i
      from 6
      to 8 by 2
      sum i)
```

from
```
(loop for i
      from 6
      to 8
      sum i)
```

to
```
(loop for i
      from 6
      to 8
      sum i)
```

always
```
(loop for i
      in '(0 2 4 6)
      always (evenp i))
```

and
```
(loop for x
      below 5
  when (= x 3)
  do (print "do this")
  and
  do (print
    "also do this")
  do (print
    "always do this"))
```

maximize*
```
(loop
   for i
   in '(1 2 3 2 1)
   maximize i)
```

then
```
(loop repeat 5
      for x = 10.0
      then (/ x 2)
      collect x)
```

upfrom
```
(loop for i
      upfrom 6
      to 8
      sum i)
```

upto
```
(loop for i
      from 6
      upto 8
      sum i)
```

never
```
(loop for i
      in '(0 2 4 6)
      never (oddp i))
```

else
```
(loop for i
      below 5
      if (oddp i)
      do (print i)
      else
      do
   (print "w00t"))
```

append*
```
(loop for i
      below 5
      append
    (list 'Z i))
```

downfrom
```
(loop for i
      downfrom 10
      to 7
      do (print i))
```

downto
```
(loop for i
      from 10
      downto 7
      do
   (print i))
```

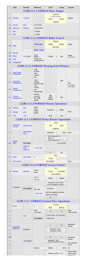
thereis
```
(loop for i
      in '(0 2 4 555 6)
      thereis (oddp i))
```

end
```
(loop for i
      below 4
      when (oddp i)
      do (print i)
      end
      do (print "yup"))
```

nconc*
```
(loop for i
      below 5
      nconc
    (list 'Z i))
```

(source https://www.reddit.com/r/Common_Lisp/comments/p6ookm/i_made_a_high_quality_copy_of_periodic_table_of/)

Or this small clip of a huge table for format;



(source https://www.hexstreamsoft.com/articles/common-lisp-format-reference/clhs-summary/)

Loop

Powerful, ugly if you think Lisp is pretty, loved, hated, and often misunderstood, the loop macro is all of that and likely more. There are times when loop can solve issues with simplicity that would be considerably more difficult without, and the reason worth knowing. This tutorial will attempt to familiarize you enough with loop to let you use that periodic table of loop to make more complex loop programs when needed[58].

A simple start:

```
(loop
  [ set up the loop(s) ]
  do
  [ what to do each time]
)
```

The keyword "do" can be written "doing". The various keywords mentioned below are not written as lists, i.e. not before or after parenthesis.

For style loops:

- for N from 1 to 5
- for N from 1 to 5 by 2
- for N below 5
- for N in '(1 2 3 4 5)
- for N in '(1 2 3 4 5) by #'cddr ; skip 1 cdr, skip 2 cddr, etc
- for N across #(1 2 3 4 5) ; for arrays/vectors[59]
- for N from X downto Y by Z
- for N from X upto Y by Z
- for N from X below Y ; step up Y - 1
- for N from X above Y ; step down Y + 1

e,g, `(loop for y from 1 to 5 do (print y))`

While/Until style loops

- for N = X then Z while (condition) do (body) ; Z is increment like (1+ N)
- for N = X then Z do (body) until (condition)

e.g. `(loop for y = 3 then (1- y) do (print y) until (= y 0))`

Repeat

[58] It is interesting to note the most common keyword in loop "collect" is not in the chart.
[59] In the next tutorial we will cover arrays/vectors and hash tables and also their use in loop.

- repeat x-times

e.g. `(loop repeat 5 for y = 1 then (1+ y) for x = 10 then (1- x) do (print (list x y)))`

Conditions
Conditions can be used to add additional do (body)s based on conditions. (end is only needed if follow by another do body that executes not related to condition follows)

- if (condition x) do (body) else do (body) end
- when (condition x) do (body) end
- do (body) until (condition x) end

e.g.
```
(loop for y from 1 to 5 if (= y 3) do (print "---") end do (print y))
```

True/False Conditions
These conditions are used instead of a do (body) to answer true or false about a loop.
- always (condition) - every turn of loop true then returns true
- never (condition) - never happened that condition true then returns true
- thereis (condition) - there is a time it turned true then returns true

Initially / Finally
Execute some code before or after the loop
- initially (body)
- finally (body)

e,g,
```
(loop initially (print "start")
  for y from 1 to 5 do (print y)
  finally (print "done"))
```

Advanced

So now that we have some loop basics, now let's get a bit deeper:

Destructing

Given a list a for loop can do a destructing-bind, which means for each pass of the list can bind multiple results:

e.g.
```
(loop for (a b) in '((1 2) (3 4) (5 6)) do (print (list a b)))
```

Collection

There are seven data collection keywords that can be used to set the return value of the loop (all work if write with ing as well):

- collect a-loop-var - every turn of loop a-loop-var's value to a list
- Append (form => list) - append list to collected result
- nconc (form => list) - concatenate to collected result
- count a-loop-var - return count of a-loop-var generated
- sum a-loop-var - generate a sum
- maximize a-loop-var - the max value
- minimize a-loop-var - the min value

e.g.
```
(loop for (a b) in '((1 2) (3 4) (5 6)) collect a)
=> (1 3 5)
(loop for (a b) in '((1 2) (3 4) (5 6)) collect (list a b))
=> ((1 2) (3 4) (5 6))
(loop for (a b) in '((1 2) (3 4) (5 6)) collect b do (print (list a
b)))
(1 2)
(3 4)
(5 6)
=> (2 4 6)
(loop for (a b) in '((1 2) (3 4) (5 6)) append (list a b))
=> (1 2 3 4 5 6)
(loop for (a b) in '((1 2) (3 4) (5 6)) count a)
=>3
```

Variables
Loop variables can be added using:

```
[collection] into new-loop-var
```

e.g.
(loop for x from 5 to 10 sum x into z do (print z))

and they can be added directly using:

```
with new-loop-var
```

That wasn't too bad :).

Format

Format is the universal data to string function in Lisp. For consoles/non-graphical paper printers it does much more, but we will not be spending time beyond a few basics used often in format strings for graphical content.

```
(format stream control-string data1 data2…)
```

Stream if "t" outputs the final output string to the *standard-output* stream.
If stream is nil then form evaluates to the output string.

Basics

The number of data arguments is based on the control-string directives.

Directives start with a tilde ~ and the tilde twice ~~ is how to quote it. If no directives are in the control-string then it is returned as the output string:

(format t "Hello World") is the equivalent of (princ "Hello World")

The directive ~% is equal to adding (terpri),i.e. a new line. The ~& is an alternative to ~% that prints a new line only if not already on a new line like (fresh-line).

(format t "Hello World~%") is equivalent to (princ "Hello World")(terpri)

~A allows for:

```
(format t "Hello ~A, Lisp is cool!~%" "David")
Hello David, Lisp is cool!
```

~A is sort of the universal argument converter, as it takes any Lisp type.

```
(format t "~A ~A ~A ~A" "David" 'David :David 123.45)
David DAVID DAVID 123.45
```

```
(let ((r (format nil "~A ~A ~A ~A" "David" 'David :David 123.45)))
  (reverse r))
=> "54.321 DIVAD DIVAD divaD"
```

Lists

Format's ability to process lists often comes in handy in modern software. The ~{ directives ~} list directive allows formating for each element in list:

```
(format t "~{Element of List: ~A~%~}" '(1 2 3 4))
Element of List: 1
Element of List: 2
Element of List: 3
Element of List: 4
```

Further Information

I would suggest looking at these examples for numerics and tabulation:
http://www.lispworks.com/documentation/lw50/CLHS/Body/22_ck.htm

The CLHS pages on Format
http://www.lispworks.com/documentation/lw50/CLHS/Body/22_c.htm

A Reference Chart by Jean-Philippe Paradis (Hexstream)
https://www.hexstreamsoft.com/articles/common-lisp-format-reference/clhs-summary/

Hash Tables and Arrays - Part 9

I want a new drug, one that won't make me sick
One that won't make me crash my car
Or make me feel three-feet thick
I want a new drug, one that won't hurt my head
One that won't make my mouth too dry
Or make my eyes too red
– Huey Lewis and the News

Introduction

The Drug Enforcement Administration (DEA) warns that hash is the most potent and concentrated form of cannabis. Similarly a Lisp hash is one of the most potent "container" types. It creates a mapping between a key object and a value object using a hash value under the covers to help speed up indexing. Lisp offers other alternatives to the hash (two objects with a bowl between them[60]), association lists (two objects that are acquaintances sharing the same cons) and the p-lists (objects in a line with just a little space between them).

The Three Amigos - All Guilty by Association

Amigo 1 - The Association List - The a-list

An association list is a list of cons objects. A cons object is a pair of objects the first called the car and the second the cdr, it is entered into the REPL using a (cons 1 2) or the short hand dot notation of '(1 . 2).

Create an association[61]:

```
(defparameter *alist* (list (cons 1 2) (cons 3 4)))
```

Append an association (nondestructive):

```
(acons 5 6 *alist*) => '((1 . 2)(3 . 4)(5 . 6))
```

[60] I am in no way condoning drug use, the overdose of hash resulted in perl's syntax, we can't let things like that happen again.
[61] We can not use the short form for lists: `(defparameter *alist* '((1 . 2)(3 . 4)))` as the literal form is a constant and we modify it destructively in our examples.

Look up association by key:

```
(assoc 3 *alist*) => (3 . 4)
(car (assoc 3 *alist*)) => 3
(cdr (assoc 3 *alist*)) => 4
```

Replace value:

```
(setf (cdr (assoc 3 *alist*)) 5)
*alist*
=> ((1 . 2) (3 . 5))
```

Reverse lookup by value:

```
(rassoc 2 *alist*) => (1 . 2)
```

Both assoc and rassoc return the first occurrence or the key or value.

Using LOOP :

```
(loop for (k . v) in *alist* do (format t "k=~A v=~A~%" k v))
k=1 v=2
k=3 v=4
```

Amigo 2 - The Property List - The p-list

A property list is a list where odd elements are keys and even elements are values.

Creating a p-list:

```
(defparameter *plist* (list 'k1 'v1 'k2 'v2 'k3 'v3))
```

Looking up a key:

```
(getf *plist* 'k2)
=> V2
```

Removing a key/value pair

```
(remf *plist* 'k2)
=> T
*plist*
```

```
=> (K1 V1 K3 V3)
```

Replacing a value

```
(setf (getf *plist* 'k3) 3)
=> 3
*plist*
=> (K1 V1 K3 3)
```

NOTE: A word about **setf**. This pattern of using setf and getf (or another "get" operator to find the field to set) is very common in Common Lisp. Think of getf as "find and get the field" and once you have the field you can "set the field".

Using LOOP :

```
(loop for (k v) on *plist* by #'cddr do (format t "k=~A v=~A~%" k v))
k=K1 v=V1
k=K3 v=3
```

There is an interesting feature of Lisp that every symbol has a p-list of its own. It is accessed by using SYMBOL-*plist*: for example

```
(getf (symbol-plist 'some-symbol) 'some-key)
```

or

```
(setf (getf (symbol-plist 'some-symbol) 'some-key) 'some-new-value)
```

Amigo 3 - The Hash Table

For anything requiring performance, more functionality, or the key is not a symbol (like a string) choose a hashtable over a-lists and p-lists.

Creating a hash table[62]:

```
(defparameter *hash* (make-hash-table :test #'equal))
```

The test keyword argument by default is eql which works for symbols and numbers, but will not work for strings. If you want the keys to not care about case, use instead #'equalp.

Adding key/value pairs (or replacing if key exists):

[62] If writing a CLOG app, CLOG includes a thread safe macro version `make-hash-table*` and you should use it instead, you will thank me later!

```
(setf (gethash "key1" *hash*) "value1")
(setf (gethash "key2" *hash*) "value2")
(setf (gethash "key3" *hash*) "value3")
```

Getting the value of a key:

```
(gethash "key1" *hash*)
```

Removing a key/value pair:

```
(remhash "key2" *hash*)
```

Empty hash table:

```
(clrhash *hash*)
```

Using Loop :

```
(loop for key being the hash-keys of *hash* collect key)
(loop for value being the hash-values of *hash* do (print value))
(loop for key being the hash-keys of *hash* using (hash-value value)
  collect (list key value))
```

Arrays and Vectors

Arrays are an alternative way to store data indeced by number(s).

Creating an array:

```
(defparameter *array* (make-array '(9 9 9) :initial-element 1))
```

In this case *array* has three dimensions, each can be from 0 to 8 and each was set to 1.

Accessing a cell:

```
(aref *array* 1 2 3)
```
=> 1

Returns the value of the element at cell 1 2 3.

Changing the value of a cell:

```
(setf (aref *array* 1 2 3) 2)
```

Vectors are single dimensional arrays that offer some additional operators.

Creating a vector:

```
(vector 1 2 3)
```

or with the shorthand

```
#(1 2 3)
```

Access is using AREF like an array.

Using Loop :

(loop for n across #(1 2 3) do (print n))

Mapping - Part 10

So I wonder where were you?
When all the roads you took came back to me
So I'm following the map that leads to you
Ain't nothing I can do
The map that leads to you
Following, following, following to you
Ain't nothing I can do
The map that leads to you
Following, following, following
– Maps - Maroon 5

Introduction

From the time we are small children we are taught to form lines. We line up to pay, line up to get in and to get out, line up for fire drills, etc. Lines, sequences, are efficient ways to process information. The `map` function in Lisp iterates across any type of sequences, other forms of map functions are further tuned for lists, and apply a function to them.

Map

Once we illustrate `map`, the other list only map functions will be easy.

```
(map 'list (lambda (item) (format nil "'~A'" item)) '(a b c))
=> ("'A'" "'B'" "'C'")
```

With map we collect the results from the lambda expression in a `'list` type, we could have used other sequence types like a vector as well:

```
(map 'vector (lambda (item) (format nil "'~A'" item)) '(a b c))
=> #("'A'" "'B'" "'C'")
```

The lambda expression's parameters (in this case just one, `item`) match the number of sequences being mapped over (in this case just one, `'(a b c)`) , here is an example with two (the shortest sequence is maximum iterations):

```
(map 'list (lambda (item1 item2) (format nil "'~A' ~A" item1 item2))
  '(a b c) #(1 2 3 4))
=> ("'A' 1" "'B' 2" "'C' 3")
```

You can also replace the return type with nil and the results of the lambda expression are discarded.

The List Versions of Map

mapcar

On each iteration item is the **car** of the next list element:

```
(mapcar (lambda (item) (format nil "'~A'" item)) '(a b c))
=> ("'A'" "'B'" "'C'")
```

maplist

On each iteration item is the **cdr** from the next list element:

```
(maplist (lambda (item) (format nil "'~A'" item)) '(a b c))
=> ("'(A B C)'" "'(B C)'" "'(C)'")
```

mapc and mapl

Are identical to mapcar and maplist but the maps evaluate to nil

mapcan and mapcon

Are identical to mapcar and maplist but they are collected using nconc.

Seek and You Will Find

find

`find` an element in a sequence:

```
(find 3 '(1 2 3 4))
=> 3
(find 5 '(1 2 3 4))
=> nil
```

`find-if` and `find-if-not` test each element using a predicate function, i.e. one that returns nil if false or any other value if true.

```
(find-if #'oddp '(2 4 3 5))
=> 3
```

It is possible to use a lambda expression with find-if or find-if-not and have a short circuit map as well:

```
(find-if (lambda (item) (print item)(oddp item)) '(2 4 3 5))
2
4
3
=> 3
```

position

`position` is like find but returns where position in the sequence (0 based) where the element is found.

```
(position 3 '(1 2 3 4))
=> 2
```

`position-if` and `position-if-not` can be used exactly as find-if/find-if-not but evaluating where element was found.

Structures and Classes - Part 11

Pain!
You made me a, you made me a believer, believer
Pain!
You break me down and build me up, believer, believer
Pain!
Oh, let the bullets fly, oh, let them rain
My life, my love, my drive, it came from...
Pain!
You made me a, you made me a believer, believer
– Believer - Imagine Dragon

Introduction

The key to large systems is modules with clearly defined interfaces/protocols that are not tied to how they are implemented. Objects in many languages were/are the only means to achieve this and so the concept of objects became intertwined with modules. Over the course of time language designers realized that the Object model was not enough and added templates, namespaces and other "refinements", but due to the static nature of these languages and their "standards" redone every few years, their object models could not be changed or enhanced without becoming a new language. **Pain!**

Common Lisp has always had the advantage of being the "programmable language" and many object implementations were used before the standardization of CLOS (the Common Lisp Object System). **You break me down and build me up, believer, believer**

Common Lisp breaks down every part of the object oriented programming separately (Inheritance, Encapsulation, Polymorphism, and Data abstraction) allowing you to assemble the parts to match more closely the domain's problems and not force any model of a solution and in that regard there is no CLOS, there is only Common Lisp and its diverse dictionary of operators[63]. **Oh, let the bullets fly, oh, let them rain**

The Old Way

The first adoption of the general concept of objects in Lisp (sometimes called object based vs object oriented programming at his level) is the abstract data type (ADT) / aggregate type /

[63] Common Lisp's greatest power is in diversity while still maintaining homoiconicity. It is why Common Lisp is still the birthplace of most new paradigms in programming, and when not the birthplace, it is the place where they come to crystalize and reach their peak. Now if only humans understood no two people think alike, but they all feel equally.. **Pain! You made me a, you made me a believer, believer**

struct / structure / record type with single inheritance (or with out inheritance and emulating it)[64]. In Common Lisp this was implemented with `defstruct`.

It may be the case that defstruct has some "optimization" as far as speed (the main is speed to type it in) over using the 'object oriented" defclass, You mostly lose functionality for a gain that means nothing in the application space with current compilers and in the systems space, are rarely going to use defstruct anyways if you need those gains. We are still going to cover the very basics of defstruct but I don't recommend using defstruct unless required by something legacy (all ffi code is by definition legacy, even if new foreign code, since Lisp is way ahead).

```
(defstruct my-record
  "A sample struct"
  first-name
  last-name)
```

The `defstruct` macro does a number of things, a few for this tutorial's purpose:

1. A definition for an aggregate data structure in memory. In the above example with 2 slots.
2. A constructor make-struct-name (with keywords for each slot, initargs, to set values on creation or they are set to nil), `(setf an-instance (make-my-record :first-name "David" :last-name "Botton"))`
3. A copy constructor copy-struct-name, `(setf new-instance (copy-my-record an-instance))`
4. For each member/component/slot[65] of the structure an accessor is created (struct-name-element). `(my-record-first-name an-instance)` returns the value of the slot and `(setf (my-record-first-name an-instance) "name")` to set the value of the slot.
5. Add it to the type hierarchy, ie. `(typep an-instance 'my-record)` is true of `(type-of an-instance) => MY-RECORD`

`defstruct` allows a number of options for the entire struct (like single inheritance :inclure) or individual slots (like types for validation if the compiler supports it :type), but if you need those or other options, you really should be using `defclass`.

What no dot notation!

[64] All concepts of OO can easily be added to any language using macros/templates, pointers and other techniques and in other languages I've had to use those techniques, it is always nicer to have a language that can morph itself like Lisp instead.
[65] I know a host of other more obscure names as well… slot is the Lisp lingo

No, defstruct (and defclass) don't need no stinkin' dots. Because encapsulation is not part of the object in Lisp, remember Lisp has packages so there is no need to combine the concept of encapsulation and defstruct and so limit structures, and because of the homoiconicity concept.

Here is something that "looks" closer to object languages (and not saying you should do this but cool to try[66]):

```
(defpackage :my-record
  (:use :cl) (:export make-object display))
(in-package :my-record)
(defstruct object
  first-name
  last-name)
(defun display (object)
  (format t "~A ~A~%" (object-first-name object)
                      (object-last-name object)))
```

Now let's test our new OO struct/class :)

```
(in-package :cl-user)
(defvar an-instance)
(setf an-instance (my-record:make-object :first-name "David"
                                         :last-name "Botton"))
(my-record:display an-instance)
```

Before going on to `defclass`, let's see how OO `defstruct` is:

The four traits of OOP

1. Inheritance - can create child classes of a struct by passing it :include and adding a :use with its parent struct - CHECK
2. Encapsulation - that is provided by placing a defstruct in its own package - CHECK
3. Polymorphism - "use of a single symbol to represent multiple different types" - CHECK
4. Data abstraction - that is what the struct does - CHECK

So actually Common Lisp was an OOP language from the start before CLOS (and that is without even throwing in techniques with closures, etc.)

The CLOS Way

[66] This separation of encapsulation and using an ADT in this way is exactly what Grady Booch uses in his first OO book "Object oriented design with applications" that uses Ada 83.

So the official path to object oriented programming in Common Lisp is using `defclass`, `defmethod`, and `defgen`. If encapsulation of data and methods is called for, arguably `package` as well. What these bring to the table are direct language support for multiple inheritance, refined control of constructors and accessors, dynamic dispatching and multiple dispatching.

defclass

Using the same structure of my-record:

```
(defclass my-record ()
  ((first-name
    :accessor first-name
    :initarg :first-name
    :initform "")
   (last-name
    :accessor last-name
    :initarg :last-name
    :initform "")))
```

This defclass macro will create (amongst much other hidden mojo, some described later) :

1. A definition for an aggregate data structure in memory. In the above example with 2 slots.
2. Allow initialization of new instances using `make-instance`, if `:initarg` specified allow for initialization with `make-instance`, and if not is unbound or the result of `:initform` if specified e.g. `(setf an-instance (make-instance 'my-record :last-name "Botton"))`
3. There is no copy constructor generated as there is nothing simple about copying objects, you need to decide what "copy" means to your object.
4. Create if specified an `:accessor` (or `:reader` or `:writer`) for each slot, e,g. `(setf (first-name an-instance) "David")`
5. Add it to the type hierarchy, ie. `(typep an-instance 'my-record)` is true of `(type-of an-instance) => MY-RECORD`

So defclass, so far, has added a more configurable interface to creating an object and removed the copy constructor. You can also inherit as mentioned from multiple declass definitions, e.g.

```
(defclass phone-record ()
  ((phone
    :accessor phone)))

(defclass employee-record (my-record phone-record)
  ((title
    :accessor title)))
```

The class employee-record has 4 slots, first-name, last-name, phone and title. Only first-name and last-name can be set on initialization. The other two must be set after.

defmethod

`defstruct` in order to associate a function with the struct/class, required that we use the package technique above and for inherited classes to `:use` the parent package. `defmethod` removes that requirement by directly associating the function (called a "method" in OO parlance and so the name defmethod) and the class. It accomplishes this by restricting a method to be invoked with an argument of the class or its descendants. (We will explain what this restriction "buys" you in defgeneric).

So to limit what a function can do and get to call it a method, we use defmethod and "type" the parameter. e.g.

```
(defmethod setup-new-employee ((employee employee-record) title
phone)
  (setf (title employee) title)
  (setf (phone employee) phone))
```

So now:

```
(setf an-instance (make-instance 'employee-record
                   :first-name "George"
                   :last-name "Forist"))

(describe an-instance)
#<EMPLOYEE-RECORD {100385B5E3}>
  [standard-object]

Slots with :INSTANCE allocation:
  PHONE                         = #<unbound slot>
  FIRST-NAME                    = "George"
  LAST-NAME                     = "Forist"
  TITLE                         = #<unbound slot>

(setup-new-employee an-instance "Boss Man" "954-555-1212")

(describe an-instance)
#<EMPLOYEE-RECORD {100385B5E3}>
  [standard-object]
```

```
Slots with :INSTANCE allocation:
  PHONE                          = "954-555-1212"
  FIRST-NAME                     = "George"
  LAST-NAME                      = "Forist"
  TITLE                          = "Boss Man"
```

Quick Summary So Far

We have a class (employee-record) that inherits from 2 super classes (my-record and phone-record), so has 4 slots that are exposed by 4 accessor methods and 1 regular method `setup-new-employee`. The encapsulation is in the package, so we can argue that all 4 of these methods and even the classes are all "private". We would need to export the class symbol, export the accessor method symbols and export the `setup-new-employee` if we wanted the equivalent of other language modes.

defgeneric

A generic function serves a few purposes we will discuss soon, what a generic function is though is a regular function that decides what method(s) will be called based on the arguments used, i.e. they provide the plumbing for dynamic dispatch[67].

Dynamic dispatch makes it possible for the same function to be called at runtime and dispatched to the appropriate objects method. In Lisp this can be spelled out (and customized of course) with a defined generic function, as opposed to the implied generated ones.

We actually created 9 generic functions under the covers so far. Accessors are setfable methods so each accessor created 2, and our `setup-new-employee` 1.

While it is possible to write an OO application and never define the generic functions, doing so helps define clear interfaces/protocols for an object that are meant to be public and to augment Lisp's only other built in way to specify modularity in code, the export on a package.

A defgeneric is written out like a function with no body and of no type typing of parameters. e.g.

```
(defgeneric setup-new-employee (employee-object title phone))
```

[67] Dispatch is the real concept they were implying with the claim the polymorphism is an integral part of OOP

Conclusion

At this point in tutorial form we have covered function to object methods and everything in between (there is more to Lisp for sure and more detail in all) but we have enough to create significant functional applications and a picture is worth a 1000 words according to Kodak. We will follow this tutorial with a companion project that will exercise what we have covered using CLOG for our GUI and then return to cover some additional important parts of Common Lisp.

Macros - Part 12

expand your mind to understand we all must live in peace to earth
extend your hand to help the plan
of love to all mankind on earth
expand your mind
expand your mind
— Lonnie Liston Smith And The Cosmic Echoes

Introduction

Lisp macros are a mind bending tool and learning them, a mind expanding experience.

What is a macro?

A macro is a **function** that outputs source code which is then generally executed.

```
(defmacro my-mac ()
  (print "hello world"))

(my-mac)
"hello world"
=> "hello world"

(defun my-func ()
  (print "hello world"))

(my-func)
"hello world"
=> "hello world"
```

The result of either is the same, the side effect of "hello world" printed and both evaluate to the string "hello world". The difference doesn't become apparent until you output code:

```
 (defmacro my-mac ()
  '(print "hello world"))

(my-mac)
"hello world"
```

```
=> "hello world"

(defun my-func ()
  '(print "hello world"))

(my-func)
=> (PRINT "hello world")
```

The results now are very different. The macro version first evaluated to a list (exactly as the function version) and that list is then evaluated as code.

In a nutshell that is what lisp macros are.

Writing Code in Code

Since code has the same form as data (homoiconicity) all of the operators we would use for creating and manipulating lists can be used in our macros, along with the entire Lisp language. The important thing to recall is that the output of our macro (its "expansion") in the end will itself be executed.

```
(defmacro example ()
  (let ((stack nil))
    (push 'print stack)
    (push "hello world" stack)
    (reverse stack)))

(example)
"hello world"
=> "hello world"
```

In tutorial 6 we learned about using a single quote shortcut to produce a list. There is an additional shortcut using a backquote that works like the single quote but has a backdoor using a comma ',' to allow unquoting the next item and evaluating it.

```
(defun hello (name)
  `(print ,name))

(hello "David")
=> (PRINT "David") ; a list with two elements 'PRINT and a string
```

or

```
(defun hellof (name)
  `(print ,(format nil "Hello ~A" name)))

(hellof "David")
=> (PRINT "Hello David")
```

So as a macro:

```
(defmacro hellom (name)
  `(print ,(format nil "Hello ~A" name)))

(hellom "David")
"Hello David"
=> "Hello David"
```

In addition to the idea that a function is evaluated and a macro's output is evaluated, there is an affect on the input, ie. the parameters as well. Arguments to macro are not evaluated.

Using the function version:

```
(hellof David)
The variable DAVID is unbound.
   [Condition of type UNBOUND-VARIABLE]
```

Using the macro version:

```
(hellom David)
"Hello DAVID"
=> "Hello DAVID"
```

The argument is evaluated in the function version and the symbol doesn't exist. In the macro version the argument is just passed in unevaluated as if it was (quote name), i.e. data.

The other backquote backdoor was the comma-splice:

```
(defmacro splice-in (to-splice)
  `(print '(1 2 3 ,@to-splice 4 5)))

(splice-in (9 9 9))
(1 2 3 9 9 9 4 5)
=> (1 2 3 9 9 9 4 5)
```

Given a list it is spliced into another list.

When to use macros?

1. Macros allow new operators that do not evaluate arguments
    a. For syntactic sugar
    b. To modify arguments
    c. To short circuit arguments
2. Macros allow for compile-time calculation
    a. Conditional compilation
    b. Compile-time optimizations
3. Code generation
    a. Reduce code duplication
    b. Link models and views
4. And more

Where from here?

The goal was to learn the initial mechanics of macros. Where to go from here…. When you are ready, you will know.

Exceptions and Conditions - Part 13

I woke up this mornin' with the sundown shinin' in
I found my mind in a brown paper bag within
I tripped on a cloud and fell-a eight miles high
I tore my mind on a jagged sky
I just dropped in to see what condition my condition was in
– Kenny Rogers and The First Edition

Introduction

Lisp is not like most languages; why should its exceptions be like other languages'. In most languages, no self respectin' programma' would consida' usin' exceptions for mo-wr than exceptional situation-is. Nope, in lisp we 'ave an entire condition system[68].

Macros let Lisp **program itself** to make decisions **at compile time**, the condition system lets Lisp **program itself** to make decisions **at or post run time**.

I've Fallen and I Can't Get Up

The most basic use of the condition system is the try and catch of thrown "exceptions".

Step 1 - Make a Condition

Create an exception class to "throw", i.e. a condition to signal in Lisp parlance, using:

```
(define-condition i-am-falling (error)())
```

In this case we made a condition class `i-am-falling`, a child of the error condition class, with no additional slots. We can signal our condition anywhere in our code:

```
(error 'i-am-falling)
```

Running that from the REPL will give us a message with the condition class that was thrown, i.e. `i-am-falling`, restart possibilities (retry and abort in our case) and a backtrace.

[68] Special thanks to https://lingojam.com/CowboyTalkTranslator

(Later we will learn what a restart is but it is just too hard to hold back the lisp side of the force. The following will sometimes work and sometimes fail:

```
(dotimes (n 100)
  (when (eql (random 100) 50)
    (error 'i-am-falling :message (format nil "from level ~A" n))))
```

When it fails (as it will most times) the debugger gives you the option to abort, or you can retry. If you **retry**, our entire dotimes expression is "retried" from 0. It has been resurrected and born again. Go try that with the dark side of the force!)

`error` is like `make-instance` for conditions (actually `make-condition` is, `error` also signals the condition after making the condition). So we can add slots with initargs to our condition:

```
(define-condition i-am-falling (error)
  ((message
    :initarg :message
    :accessor message)))
```

and when we signal the condition creating our i-am-falling object we can supply initargs:

```
(error 'i-am-falling :message "and I can't get up!")
```

If you evaluated that code though, nothing really changed except the backtrace, ie. **Where is my message?!** Ah to do that we need to add a ":report" to our condition:

```
(define-condition i-am-falling (error)
  ((message
    :initarg :message
    :accessor message))
  (:report (lambda (condition-object stream)
             (format stream "HELP! I am falling ~a.~&"
                     (message condition-object)))))
```

`condition-object` is the object we created an instance of when we signaled with `error` and stream is the debugger's output stream. Our `message` slot is accessed by our accessor just like any other object's slot in Lisp.

Step 2 - Get ready to catch

Signaling conditions is a great way to "throw" your app into the debugger, but most of us want to "handle" conditions instead.

So to "try" a piece of code, we wrap it in a `handler-case`[69], then if a condition is "thrown", the `handler-case` will "catch" it.

```
(handler-case
     (error 'i-am-falling :message "I can't get up")
  (i-am-falling (condition-object)
    (format t "I can handle \"~A\"." (message condition-object))))
```

`handler-case` does not have an implied progn so if needed you will have to wrap multiple forms in a progn.

A series of possible conditions can be caught, adding additional conditions:
```
 (condition-class (condition-object) body)
```
and a default for any condition can be added with
```
 (t (condition-object) body)
```
and a no-error condition also possible (result is the result of the handler-case expression)
```
  (:no-error (result) body)
```

So we can create a little game:
```
(handler-case
    (dotimes (n 100)
      (when (eql (random 100) 50)
        (if (eql (random 1000) 1)
            (error "SPLAT!") ; some generic error condition
            (error 'i-am-falling :message (format nil "from level ~A" n)))))
  (i-am-falling (condition-object)
    (format t "I fell ~A." (message condition-object)))
  (t (condition-object)
    (format t "Oh MY: ~A" condition-object))
  (:no-error (result) (declare (ignore result)) (print "MADE IT!")))
```

Each time you run, you either fall from a level, make it or .. SPLAT!

You Can Do It! Yes You Can!

Restarts are a way to program in possible recovery situations to conditions including interacting with the user for other possible solutions.

Let's use restarts for our condition game.

```
(defun climb ()
```

[69] The -case portion of handler-case is from an advanced operator `typecase` not discussed in "The Tutorial" series, that allows selection based on type as is the case here.

```
    (dotimes (n 100)
      (when (eql (random 100) 50)
        (if (eql (random 1000) 1)
            (error "SPLAT!")
            (error 'i-am-falling :message (format nil "from level ~A" n)))))
  (print "We Made It!"))

(loop
  (restart-case (climb)
    (play-again ()
      nil)
    (done-here ()
      (print "GAME OVER")
      (return))))
```

This example so far really is not taking advantage of the idea of "restarting", but we will add on to it. First we place our climb game into its own function and give it a game loop. We wrap `climb` in a handler to capture conditions but this time it is a handler that can provide restarts.

So play by play:

1. Our game loop starts
2. We call climb
    a. If `climb` succeeds we get our "We Made It!" message and the game loop starts again.
    b. If `climb` fails a condition is signaled either a generic error or our `i-am-falling error`. `restart-case` does not differentiate what type of condition was signaled. The debugger though will report based on the condition class. `restart-case` does allow us to decide what to do. In `restart-case` so far there are two options. Each option will be displayed in debugger and you get to pick which one to take.
        i. Option 1 - play-again = trash the condition, do nothing as if all was ok, i.e. loop again.
        ii. Option 2 - done-here = trash the condition, execute the form (return), i.e. "GAME OVER" return from our loop.

When you run our little game so far and we fail to make it we get our error:

```
   HELP! I am falling from level 49.
```

But we now have a few **restarts** available:

```
 0: [PLAY-AGAIN] PLAY-AGAIN
 1: [DONE-HERE] DONE-HERE
 2: [RETRY] Retry SLIME REPL evaluation request.
 3: [*ABORT] Return to SLIME's top level.
 4: [ABORT] abort thread (#<THREAD "worker" RUNNING {10046D6C63}>)
```

The first two were provided by our game. The next 3 are tacked on by restarts available from a "higher" place, no not God, from higher places on the stack, in this case the REPL that gave life to our game loop. So choice labeled 2 is to go back in time and try the REPL command again. My parents wish sometimes they could do that to me………… The other choices labeled 3 and 4 are to give up, and not restart at all.

So far this is just abusing restarts to be our game interface :) Let's actually add some restarts, ie. ways to retry `climb` so we don't fail.

First let's make climb adjustable:

```
(defun climb (&key (levels 100) (difficulty 100))
    (dotimes (n levels)
      (when (eql (random difficulty) 50)
        (if (eql (random 1000) 1)
            (error "SPLAT!")
            (error 'i-am-falling :message (format nil "from level ~A" n)))))
  (format t "We Made It! climbing ~A levels with ~A lbs on our back.~%"
          levels
          difficulty))
```

Now let's add the ability if we fail to *turn back time* and change the levels or the difficulty on the last climb:

```
(loop
  (restart-case (climb)
    (play-again ()
      nil)
    (change-difficulty (value)
      :report "Enter a new difficulty"
      :interactive (lambda ()
                     (prompt-new-value "Please enter a new difficulty: "))
      (climb :difficulty value))
    (change-levels (value)
      :report "Enter how many levels"
      :interactive (lambda ()
                     (prompt-new-value "Please enter how many levels: "))
      (climb :levels value))
    (done-here ()
      (print "GAME OVER")
      (return))))
```

We are going to borrow a function from the *Awesome* Lisp Cookbook[70] for prompting for a new value in a restart:

[70] https://lispcookbook.github.io/cl-cookbook/error_handling.html

```
(defun prompt-new-value (prompt)
  (format *query-io* prompt)  ;; *query-io*: the special stream to make user queries.
  (force-output *query-io*)   ;; Ensure the user sees what he types.
  (list (read *query-io*)))   ;; We must return a list.
```

So now we have a much more interesting game where we can adjust any failed level and redo it with different settings.

The One More Thing Command

Now I am so happy with my "condition restart" game I want to make sure that even if someone decides to abort the game, to tell them Thank You.

To do that, after all this "trying" "catching" and *restarting*, I "finally" protect as they unwind using:

```
(unwind-protect
     (loop
       (restart-case (climb)
         (play-again ()
           nil)
         (change-difficulty (value)
           :report "Enter a new difficulty"
           :interactive (lambda ()
                          (prompt-new-value "Please enter a new difficulty: "))
           (climb :difficulty value))
         (change-levels (value)
           :report "Enter how many levels"
           :interactive (lambda ()
                          (prompt-new-value "Please enter how many levels: "))
           (climb :levels value))
         (done-here ()
           (print "GAME OVER")
           (return))))
  (print "y'all come back now ya hear"))
```

Message in a Bottle

We only talked about the error signal, but there are three signal operators:

1. `error` - we have already seen, signals the condition and if not handled :reports the condition and launches the debugger.
2. `warn` - signals the condition and if not handled it :reports the condition but does not launch the debugger

3. `signal` - signals the condition and if not handled nothing is done.

When using `define-condition` with `warning` the condition must be a child of `warning`:

```
(define-condition i-am-slipping (warning)
  ((message
    :initarg :message
    :accessor message))
  (:report (lambda (condition-object stream)
             (format stream "whoops ~a.~&"
                     (message condition-object)))))

(warn 'i-am-slipping :message "ok")
WARNING: whoops ok.
=> nil
```

When using `signal` the condition must be a child of `condition` which is the parent of both `warning` and `error` so either can be used with `signal`.

In all three cases the only major difference is what happens if a condition is not caught.

Conclusion

The Lisp way is to give you tools to micro control the language down to its most granular level and the condition system is no exception. Take a look at "The Common Lisp Condition System: Beyond Exception Handling with Control Flow Mechanisms" by Michał "phoe" Herda, a great book that works through the entire system's nuts and bolts.

Concurrent and Parallel Programming - Part 14

Deep inside of a parallel universe
It's getting harder and harder to tell what came first
I'm underwater where thoughts can breathe easily
Far away you were made in a sea, just like me
– Parallel Universe - Red Hot Chili Peppers

Introduction

First I will scare you:

Concurrent programming means writing software that potentially can run simultaneously, i.e. in **parallel** on different processor cores. In modern day computers using modern Lisp compilers and a framework like CLOG, your software projects will take full advantage of all the processors and cores available. Designing that into your code from the start maximizes your project's potential.

A **Task**[71]**,** executes inside of a **process,** your program and its address space, and if the operating system allows multiple tasks to execute, each task runs a **thread**[72] of your code[73]. Multiple threads can execute in a single task by emulating multiple tasks[74], so an application can be **multithreaded** and potentially **multitasking**.

Read the notes and get **more** SCARED.

Then boo it is not that complicated!

Read on …

[71] An operating system thread
[72] An asynchronous thread of control to create real or the illusion of parallelism, WARNING WARNING - Lispers used to call them **processes**
[73] Almost all OSes today allow multitasking.
[74] This emulation is often called **green threads** or **virtual threads** on non multitasking programs, **fibers** or **lightweight threads** when run within multi task processes. As these always emulate parallelism they never are parallel. **M:N (Hybrid Threading)** are virtual threads (fibers) that are scheduled to run on tasks and sometimes are parallel and sometimes are not even in the same application.

All you need in life, you learned in **kindergarten**

You have one room (your address space) and a lot of kindergarteners (each thread) trying to play with the same toys (your data or resource[75]). If you don't protect them, they will grab at the same time and break the toys!

Luckily most of the toys are on shelves or in boxes (they are not globals) so you don't have to protect them.

It is really fun though to play together! So what to do? Make some **rules** to protect the toys.

Rule 1 - You need parental support

Every process needs to talk to the OS to ask for support, so every Lisp implementation needs to provide an api, each invented its own. To make things simple there is a library that hides the details, bordeaux-threads. (It is already loaded if you load CLOG `(ql:quickload :clog)`)

```
(ql:quickload :bordeaux-threads)
```

Rule 2 - Create locks for your toys

The simplest way to protect your data or a resource is to only let one thread use it at a time. To do that you create a lock[76]:

```
(defvar *my-lock* (bordeaux-threads:make-lock))
```

Rule 3 - Make sure only one child gets the key at a time

To use the lock so that only one thread has access at a time you use:

```
(defvar *my-toy* 0)

(defun play-with-toy ()
  (bordeaux-threads:with-lock-held (*my-lock*)
    (incf *my-toy*)
    (format t "My toy is now : ~A" *my-toy*)))
```

[75] Like the screen for example, things can get messy, have a sentence from one thread, another half from another for example. Remember threads are running potentially in parallel.
[76] bordeaux-threads also sets a package nick name of bt, to shorten things up, but it is not my style to use nicknames unless package level.

Here we increment *my-toy* a global counter. `incf` is not an **atomic**[77] function and potentially two threads could access *my-toy* at the same time and the result would be a failure of one thread to increment *my-toy*, that is called a **race condition** (two kids race for the same toy and grab at same time). That is a very difficult bug to find as it seemingly randomly shows up since not every thread is running for the same toy at the same time. So you have to plan ahead and lock your toys up!

The lock stays locked and no one else can play with the toy until the last form is executed in the implied progn of `with-lock-held` which is also called a **critical section**.

First Day of School

First day of class all the kindergarteners arrive. They run into the classroom and each goes to grab a toy. The first one gets the key and the rest have to wait their turn.

Creating Threads - New Arrivals

To create our new arrivals we use `make-thread`:

```
(dotimes (n 10)
  (bordeaux-threads:make-thread (lambda () (play-with-toy))))
```

This creates 10 threads all running to play-with-toy. Once the lambda expression returns the thread dies. Let's modify and slow it down so we can see what is happening:

```
(dotimes (z 10)
  (bordeaux-threads:make-thread
   (lambda ()
     (let ((n z))
       (format t "Kid ~A runs~%" n)
       (sleep (random .02))
       (play-with-toy)
       (format t "I got to play ~A~%" n)))))
```

By the way, this has a race condition in it? Do you see it? It is in the output to the screen itself.

[77] An atomic function is one that executes in such a way that there is no opportunity for another thread to change anything before the function completes.

Conclusion

So threads are called **active entities**, locks are called **reactive entities**, and **passive entities** are those that are innately safe, like at atomic function[78]. There are many libraries in Common Lisp for concurrent/parallel programming, they offer different models made from combinations of these types of entities.

[78] What is atomic (or can be made to be atomic) can change from implementation to implementation of Lisp.