

TRABAJO PRÁCTICO (COMMON) LISP







[75.31] TEORÍA DE LENGUAJE PRIMER CUATRIMESTRE DE 2020

INTEGRANTES

Alumno	Padrón	Mail
del Mazo, Federico	100029	fdelmazo@fi.uba.ar
Di Santo, Javier	101696	jdisanto@fi.uba.ar
Dvorkin, Camila	101109	cdvorkin@fi.uba.ar
Secchi, Ana	99131	asecchi@fi.uba.ar

LINKS

-
-  Repositorio: <https://github.com/FdelMazo/7531-TDL>
 -  Proyecto Integrador: <https://github.com/FdelMazo/cl-aristid>
 -  Presentación: <https://youtu.be/XOIZSVfiMY>
 -  Demo: <https://youtu.be/TibZVTKYzmo>
-

ÍNDICE

1. Historia	3
1.1. Idea	3
1.2. Definición	4
1.3. Pionero	4
2. Sintaxis	5
3. Características del Lenguaje	6
3.1. Paradigma	6
3.2. Compilado/Interpretado	6
3.3. Tipado	6
3.4. Scoping	6
3.5. Recursión	6
3.6. Estructuras	7
3.6.1. Definir una estrucura	7
3.6.2. TDA Hash	7
4. Metaprogramming	9
4.1. Operadores básicos	9
4.2. Ejemplo de macro: lcomp	10
5. Lisp en la práctica	10
5.1. Estadísticas	10

1 HISTORIA

Fuentes: [1]



Figura 1: John McCarthy

"Programming is the problem of describing procedures or algorithms to an electronic calculator."

- John McCarthy

1.1 Idea

El lenguaje Lisp nace de una manera distinta a los otros lenguajes de programación. En los 50, John McCarthy propone una nueva disciplina y un nuevo campo de investigación: la inteligencia artificial. Esta disciplina está centrada en la idea de formalizar el conocimiento, es decir, está centrada en el poder escribir, de manera teórica y en términos computacionales, lo que piensa un ser humano.

En esta época no existían muchos lenguajes de programación, mucho menos lenguajes de alto nivel. Es por esto que encontrar las herramientas necesarias para los problemas que surgían era mucho más difícil de lo que nos es hoy en día. Es con este panorama complicado que McCarthy decide crear un lenguaje nuevo que le permitiera desarrollar sus ideas, inspirándose en lo que ya existiese. Tomando las ideas de Fortran e IPL-II (un lenguaje creado por colegas de McCarthy para el estudio de IA), diseña Lisp.

A diferencia de otros lenguajes (tanto del momento, como de hoy en día) Lisp no tenía sus bases fundadas en la programación imperativa, que proviene de las ideas de Alan Turing, si no que Lisp proviene de un enfoque más matemático: el Cálculo Lambda, de Alonzo Church.

El cálculo lambda es una notación matemática que permite definir todo problema computable en términos de entrada y salida. Solamente se interesa en utilizar funciones como transformadores de información. Se reciben identificadores, y se devuelve un resultado. Nunca hay un paso intermedio, ni un guardado/cargado de datos, como si lo hay en las máquinas de Turing. Esta falta de paso intermedio (que ahora conocemos mejor como estado interno) es la gran base de la programación funcional.

Al ser Lisp el primer lenguaje basado en estas ideas, se convierte en el padre fundador de la programación funcional. Algo que originalmente surge solamente de la necesidad de poder desarrollar un campo de estudio termina siendo un lenguaje que introduce desde cero un nuevo paradigma de pensamiento.

1.2 Definición

El lenguaje que diseñó McCarthy era meramente un ejercicio teórico, su único fin era usarlo como ayuda para poder continuar con sus ideas de IA. Era no mucho más que una fantasía, “imagínense si tuviésemos un lenguaje que pueda...”, o “imagínense si tuviésemos una función que pudiese evaluar cualquier expresión...”

Sin embargo, McCarthy continúa con sus estudios y sigue divulgando sus ideas al respecto. Todo esto funciona solamente en el lápiz y papel teórico hasta que McCarthy conoce a Steve Russell, un alumno suyo. Este estudiante decide intentar pasar a código máquina las funciones definidas por McCarthy, hasta que lo logra, y finalmente así, junto con la ayuda de Timothy Hart y Mike Levin, 4 años después del diseño original de John McCarthy, nace la primera implementación de Lisp, en 1962.

1.3 Pionero

Cuando McCarthy diseñó Lisp a fines de los 50, fue una salida radical de otros lenguajes existentes, el más importante de ellos siendo Fortran. Esto se debe a lo mucho que innova [2] este lenguaje en diferentes sectores del mundo de la programación.

En primer lugar, el constructor **if-then-else**, McCarthy trae por primera vez las expresiones condicionales lógicas (el `if` aritmético ya existía en Fortran). También se dan a conocer las **funciones como objetos de primera clase**, en Lisp son un tipo de dato como lo son los enteros, cadenas, etc. Tienen una representación literal, pueden ser asignadas a variables, pasadas como argumentos (parámetros). En cuanto a las **variables**, todas son efectivamente punteros. Los valores son aquellos que tienen tipos, no variables. Asignar variables significa copiar punteros, y no aquello a lo que apuntan.

Por otro lado, aparece la idea de la **recursión**, la misma ya existía matemáticamente, pero nunca en un lenguaje de programación. Lisp fue el primer lenguaje en utilizar garbage collection automático, con un diseño primitivo (no era concurrente).

Otro punto muy importante que brinda por primera vez el lenguaje es el **interprete REPL**, el cual le permite al usuario contar con un feedback inmediato, a partir de esta idea se puede programar desde abajo para arriba, compilando incrementalmente.

Por último, es interesante mencionar que en Lisp, el **lenguaje completo está siempre disponible**, por ende, no hay una distinción real entre tiempo de lectura, tiempo de compilación y tiempo de ejecución. Uno puede compilar o ejecutar mientras lee, leer o ejecutar código mientras compila, leer o compilar mientras se ejecuta el código.

2 SINTAXIS

```

1  ;; Todo en LISP se compone de symbolic expressions
2  1 ; Una s-expression puede ser un atomo -> irreducible
3  (+ 1 2) ; Una s-expression puede ser una lista -> partible
4
5  ;; Las s-expressions evaluan a valores
6  2 ; evalua a 2
7  (+ 2 3) ; evalua a 5
8  (+ (+ 2 3) 2) ; (+ 2 3) evalua a 5 -> todo evalua a 7
9
10 ;; El operador quote toma una s-expression y devuelve el codigo
11 (+ 1 1) ; evalua a 2
12 (quote (+ 1 1)) ; evalua a (+ 1 1)
13 ('(+ 1 1)) ; quote se abrevia a '
14
15 ;; El operador eval toma una s-expresion y devuelve su valor
16 (eval (+ 1 1)) ; evalua a 2
17 (eval '(+ 1 1)) ; evalua a 2
18 (eval ''(+ 1 1)) ; evalua a (+ 1 1)
19
20 ;; atom devuelve si algo es un atomo o no
21 (atom 1) ; True (el valor de la expresion 1 es un atomo)
22 (atom (+ 1 2)) ; True (el valor de la expresion (+ 1 2) es un atomo)
23 (atom '(+ 1 2)) ; Nil (la expresion (+ 1 2) es una lista)
24
25 ;; listp devuelve si algo es una lista o no
26 (listp 1) ; Nil (1 no es una lista)
27 (listp (+ 1 2)) ; Nil (la expresion evalua a 3, no es una lista)
28 (listp '(+ 1 2)) ; True (estoy hablando del codigo de la expresion,
    la lista)
29
30
31 ;; car recibe una lista y devuelve su primer elemento
32 (car (+ 1 2)) ; explota, no recibio una lista
33 (car '(+ 1 2)) ; devuelve +
34
35 ;; cdr recibe una lista y devuelve el resto (todo menos el primer
    elemento)
36 (cdr '(+ 1 2)) ; devuelve (1 2)
37
38 ;; cons crea un cons de un valor seguido de una lista
39 ;; o sea, agrega un valor al principio de la lista
40 (cons '1 '(2 3)) ; devuelve (1 2 3)
41 (cons '+ '(2 3)) ; devuelve (+ 2 3)
42
43 ;; list compone una lista de sus argumentos
44 (list 1 2 3) ; devuelve (1 2 3)
45 (list '+ 2 3) ; devuelve (+ 2 3)

```

Fuentes: [3]

3 CARACTERÍSTICAS DEL LENGUAJE

Saliendo un poco de Lisp en general, y adentrandonos en el dialecto de este trabajo práctico, damos a conocer algunas características principales de Common Lisp.

3.1 Paradigma

Common Lisp es un lenguaje multi paradigma de propósitos generales. Soporta una combinación de paradigmas de programación como programación declarativa (definir el qué sin explicar el cómo), funcional (los componentes se definen como funciones matemáticas, es determinístico, la misma entrada garantiza la misma salida, no cuenta con estado ni hay efectos secundarios y es programación de alto orden), orientada al objeto, reflexivo e imperativo. Esto le va a permitir al programador, crear programas usando más de un estilo de programación, sin estar forzado a tomar un estilo en particular.

3.2 Compilado/Interpretado

En Common Lisp, las funciones pueden ser compiladas de forma individual o por el archivo. Ya sean compiladas o interpretadas, las funciones se van a comportar de la misma forma, excepto con el comando `compiled-function-p` que verifica si la función pasada por parámetro fue compilada.

Common Lisp no es un compilador en tiempo de ejecución, sino que es necesario invocar al compilador mediante las funciones `COMPILE`, para las funciones individuales y `COMPILE-FILE`, para los archivos. El compilador puede recibir instrucciones sobre qué tan dinámico debe ser el código compilado. Luego, se cuenta con el intérprete **REPL (Read-Eval-Print-Loop)** que, como se menciono anteriormente, le permite al usuario feedback inmediato de lo que ejecuta para así poder programar desde abajo para arriba, compilando incrementalmente.

3.3 Tipado

Lisp es un lenguaje de tipado dinámico porque las verificaciones de tipo se realizan en tiempo de ejecución y las variables se pueden configurar de forma predeterminada para todo tipo de objetos.

3.4 Scoping

Cuando se habla de variables de ambito léxico, se habla de un nombre que siempre refiere a su entorno léxico local, es decir que si yo defino una variable X dentro de una función a un valor, ese será su valor adentro sin importar cualquier otro que podría tener por fuera.

En cambio, con `dynamic scoping`, se refiere al identificador asociado con el entorno más reciente. En otras palabras, se busca a la variable en el ambiente en el que la función se llama y no en dónde se define. Para que esto suceda es necesario declararla con `special`.

3.5 Recursión

La recursión en este lenguaje [4] es muy importante por diversas razones. En primer lugar, evita errores por efectos secundarios. Además, la estructura de datos de Lisp es más sencilla de utilizar con recursión. Las listas son o `nil` o `cons`. Pero además, es simplemente más elegante y limpio.

Sin embargo, hay que tener en cuenta que la solución recursiva más obvia no necesariamente es la más eficiente. Por ejemplo, la función de Fibonacci, que se define recursivamente de la siguiente forma:

1. $\text{Fib}(0) = \text{Fib}(1) = 1$.
2. $\text{Fib}(n) = \text{Fib}(n-1) + \text{Fib}(n-2)$

Al traducir esta idea a Lisp, nos encontramos con una idea poco eficiente ¹:

```
1 (defun fib (n)
2   (if <= n 1)
3     1
4     (+ (fib (- n 1))
5         (fib (- n 2)))))
```

Por otro lado, se puede resolver de forma iterativa de esta forma:

```
6 (defun fib (n)
7   (do ((i n (- i 1))
8       (f1 1 (+ f1 f2))
9       (f2 1 f1))
10      (<= i 1) f1)))
```

3.6 Estructuras

3.6.1 Definir una estructura

La macro `defstruct` en Lisp permite definir una estructura [5]:

```
1 (defstruct nombre_estructura
2   atributo1
3   atributo2
4   atributo3
5 )
```

⇒ para invocar al constructor: `make-nombre_estructura`

⇒ para acceder a un atributo: `nombre_estructura-atributo1`

3.6.2 TDA Hash

Las tablas de hash [6] son una importante estructura de datos, que asocian claves con valores de una manera muy eficiente. Los hashes son preferibles por sobre listas cuando se le da importancia a los tiempos de búsqueda, pero son más complejos lo cual hace que las listas sean las elegidas cuando solamente hay unos pocos pares clave-valor a mantener.

CREAR UNA TABLA DE HASH EN COMMON LISP

Las tablas de hash son creadas usando la función `make-hash-table`. No requiere ningún argumento obligatorio.

```
1 (defvar tabla)
2 (setq tabla (make-hash-table))
```

El argumento opcional más usado es `:TEST`, que especifica la función utilizada para testear claves iguales.

```
3 (setq tabla (make-hash-table :test 'equal))
```

¹ Repite constantemente instrucciones que ya se resolvieron

AGREGAR UN ELEMENTO A LA TABLA

Se utiliza la función `gethash`, que es la función que se encarga de devolver el elemento, en conjunto con la función `setf`

```
4 * (setf (gethash "clave1" tabla) 3) ; argumentos: la clave y el hash
5 3
```

OBTENER UN VALOR

La función `gethash` toma dos argumentos obligatorios: una clave y una tabla de hash. Devuelve dos valores: el valor que corresponde a la clave en la tabla de hash (ó `nil` en caso de que no se encuentre), y un booleano que indica si la clave fue encontrada en la tabla. El booleano es necesario ya que `nil` es un valor válido en in par clave-valor. Es decir que obtener un `nil` como primer valor de `gethash` no significa necesariamente que la clave no se encuentra en la tabla.

```
1 * (defvar tabla)
2 TABLA
3 * (setq tabla (make-hash-table :test 'equal))
4 #<HASH-TABLE :TEST EQUAL :COUNT 0 {10058B8553}>
5 * (setf (gethash "clave1" tabla) 3)
6 3
7 (gethash "clave1" tabla)
8 3
9 T
```

Guardamos NIL en el hash:

```
10 * (setf (gethash "clave2" tabla) nil)
11 NIL
12 * (gethash "clave2" tabla)
13 NIL
14 T ; T indica True, existe la clave.
```

BORRAR DE LA TABLA DE HASH

Se utiliza la función `remhash` para eliminar el par clave-valor. Es decir, la clave y su valor asociado serán eliminados por completo de la tabla. `remhash` devuelve `true` si dicho par existe, `nil` en otro caso.

```
15 * (remhash "clave1" tabla) ; elimino el par: "clave"-3
16 T
17 * (remhash "clave2" tabla) ; elimino el par: "clave2"-nil
18 T
19 * (gethash "clave1" tabla) ; trato de obtener el valor de algo que
    fue eliminado
20 NIL
21 NIL
22 * (remhash "clave3" tabla) ; trato de borrar una clave que no existe
23 NIL
```

ITERADORES DEL HASH

- `maphash`: itera sobre todas las claves de la tabla. Devuelve siempre `nil`.

```
24 ; partimos de un hash con 3 claves
25 * (defun imprimir-entrada (clave valor)
26   (format t "El valor asociado a la clave ~S es ~S~%" clave valor)
27 )
28 IMPRIMIR-ENTRADA
29 * (maphash #'imprimir-entrada tabla)
```



```

30 El valor asociado a la clave "clave1" es 1
31 El valor asociado a la clave "clave2" es 2
32 El valor asociado a la clave "clave3" es 3
33 NIL
34 *

```

- `with-hash-table-iterator`: es una macro que convierte el primer argumento en un iterador que en cada invocación devuelve un booleano generalizado que es `true` si alguna entrada es devuelta, la clave, y el valor. Si no encuentra más claves, devuelve `nil`.

```

35 * (with-hash-table-iterator (iterador tabla)
36   (loop
37     (multiple-value-bind (entrada clave valor)
38       (iterador)
39       (if entrada
40         (imprimir-entrada clave valor)
41         (return))))))
42 El valor asociado a la clave "clave1" es 1
43 El valor asociado a la clave "clave2" es 2
44 El valor asociado a la clave "clave3" es 3
45 NIL

```

4 METAPROGRAMMING

En Lisp, una macro genera y devuelve código. La forma más sencilla de pensarlo sería como una transformación de código. Cuando se llama a una macro:

1. Se arma el código en base a la definición `defmacro` de la misma.
2. Se evalúa el nuevo código en el lugar de la llamada a la macro.

Si queremos analizar el código que generaría la macro, existe la función `macroexpand` que devuelve el código generado.

A partir de esto, se pueden usar macros para simplificar y reutilizar código, cambiar el orden de evaluación, agregar más argumentos en el medio o hasta manipular la sintaxis del lenguaje.

Las macros de Lisp son macros sintácticas, y funciona al nivel del árbol de sintaxis abstracta, preservando la estructura léxica del programa original. En otras palabras, las macros en Lisp se escriben en el mismo lenguaje de Lisp, donde está disponible toda la funcionalidad del lenguaje.

4.1 Operadores básicos

```

1 (setq b 1)
2 (setq c 2)
3 (setq l '(0 0 0))
4
5 ;; backquote -> se utiliza como el quote
6 `(a b c) ; -> Lista de 3 simbolos -> (A B C)
7
8 ;; coma -> sirve para 'activar y desactivar' el backquote
9 `(a ,b ,c) ; -> b y c se evaluan -> (A 1 2)
10 `(a ,l c) ; -> b y c se evaluan -> (A (0 0 0) C)
11
12 ;; coma-at -> desempaqueta la lista (elimina los parentesis)
13 `(a ,@l c) -> (A 0 0 0 C)

```

4.2 Ejemplo de macro: lcomp

lcomp: Replicar la sintaxis de compresión de listas de Python

```

14 (defmacro lcomp (expression for var in list conditional
15   conditional-test)
16   `(let ((result nil))
17     (loop for ,var in ,list
18       ,conditional ,conditional-test
19       do (setq result (append result (list ,expression))))
20     result))
21 ; se llama de la forma
22 (lcomp x for x in (1 2 3 4 5 6 7) if (= (mod x 2) 0))
23
24 ; una vez generado y ejecutado el código devuelve
25 (2 4 6)

```

5 LISP EN LA PRÁCTICA

LISP estaba unas décadas delante de otros lenguajes de programación. Influenció a la mayoría de los lenguajes populares y altamente requeridos por la industria de la informática como Python, Ruby o JavaScript. Incluso, algunos lenguajes vienen con features con los que LISP ya contaba 39 años atrás. (Expresiones lambda por ejemplo).

Sin embargo, LISP no es tan exitoso como uno puede esperar de un lenguaje con toda esa funcionalidad superior. Lisp sirve mucho mas por el lado de tomar sus conceptos y meterlos en los lenguajes. Por ejemplo Java, que es casi puramente orientado a objetos, ahora admite programar en funcional (Java 8), JavaScript nace de Scheme, Python cuenta con numerosas funciones de Lisp como Map y Reduce.

En el artículo "Who Cares About Functional Programming?" [7], Thomas Bandt lista unos puntos acerca de "que se puede hacer para lograr que la programación funcional sea mas atractiva a la audiencia" y nota que la programación funcional cumple la mayoría de dichos puntos, a excepción de dos. Uno era la necesidad de entrenamiento; en la programación funcional hace falta aprender nuevas técnicas y "desaprender otras", pero hoy en día existen muchos metodos novedosas de como alcanzar esto.

Sin embargo, el otro factor, y probablemente el que mas pesa y la pieza que falta, es la necesidad de un "killer app": A los usuarios les va a llamar más un lenguaje, si les deja hacer algo de manera conveniente y eficaz, que de otra manera seria muy difícil de lograr. Como todas las nuevas tecnologías, los lenguajes funcionales tienen que buscarse una aplicacion que sorprenda para sobresalir en el negocio. Y a Lisp le faltó esto.

5.1 Estadísticas

STACK OVERFLOW  [8]

- ⇒ En cuanto a popularidad de LISP en preguntas dentro de Stack Overflow, vemos en la figura 2 que Common Lisp compite contra otros dialectos, pero Clojure es el más utilizado.
- ⇒ De todas formas, toda la familia de LISP es un porcentaje muy chico, en comparación a otros lenguajes.

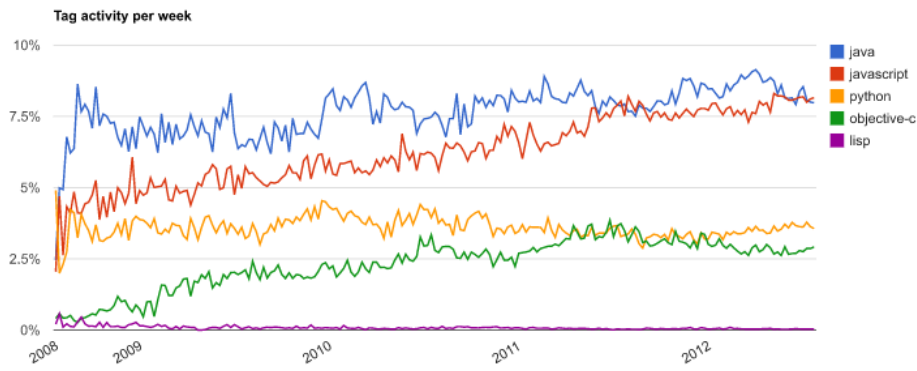


Figura 2: Porcentaje de preguntas relacionadas a Lisp contra otros lenguajes

GITHUB [9]

⇒ Podemos ver en la figura 3 que la cantidad de pull requests de Common Lisp anuales son muy bajos, incluso contra otros dialectos y lenguajes.

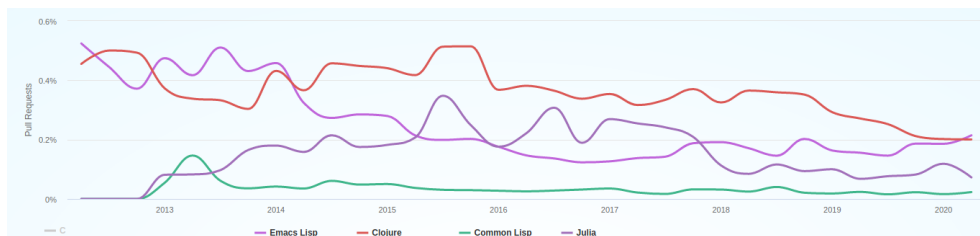


Figura 3: Porcentaje de Pull Requests en un año

HACKERNEWS [10]

⇒ Aunque Lisp no es un lenguaje que se usa mucho en la práctica, vemos en la figura 4, que sí hay un gran interés teórico por este (hay que tener en cuenta que este foro nos da una idea de que blogs o artículos se comparten de una tecnología, en vez de usos prácticos y código productivo).

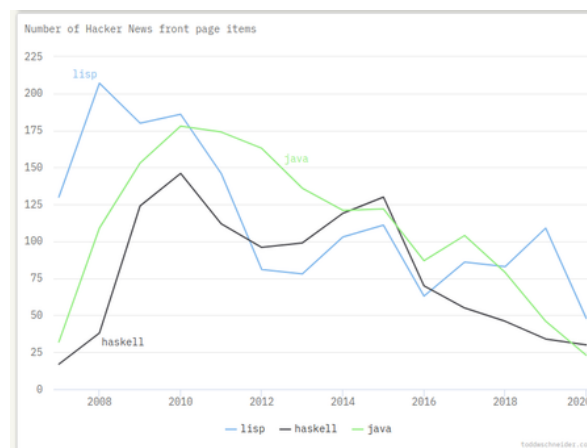


Figura 4: Cantidad de artículos anuales por lenguaje que estuvieron en la página principal del sitio

REFERENCIAS

- [1] Early LISP History (1956 - 1959) - Herbert Stoyan @
<https://campus.hesge.ch/Daehne/2004-2005/Languages/Lisp.htm>
 History of Lisp - John McCarthy @
<http://www-formal.stanford.edu/jmc/history/lisp/lisp.html>
 Revenge of the Nerds - Paul Graham @
<http://www.paulgraham.com/icad.html>
 Lets LISP like it's 1959 // LISP and the foundations of computing @
<https://youtu.be/hGY3uBHVvR4> & <https://lwn.net/Articles/778550/>
- [2] What Made Lisp Different - Paul Graham @
<http://www.paulgraham.com/diff.html>
 Influential Programming Languages, Lisp - David Chisnall @
<https://www.informit.com/articles/article.aspx?p=1671639>
- [3] LISP 1.5 Programmer's Manual @
<http://web.cse.ohio-state.edu/~routev.1/6341/pdf/Manual.pdf>
 Common Lisp HyperSpec @
<http://clhs.lisp.se/Front/index.htm>
 Learn X in Y minutes, Where X=Common Lisp @
<https://learnxinyminutes.com/docs/common-lisp/>
- [4] Simple Recursion - LEARNING LISP @
<http://shrager.org/llisp/11.html>
- [5] LISP - Structures @
https://www.tutorialspoint.com/lisp/lisp_structures.htm
- [6] The Common Lisp Cookbook, Hash Tables @
<http://cl-cookbook.sourceforge.net/ashes.html>
 LISP Hash Table @
https://www.tutorialspoint.com/lisp/lisp_hash_table.htm
- [7] Who Cares About Functional Programming? - Thomas Bandt @
<https://thomasbandt.com/who-cares-about-functional-programming>
- [8] Stack Overflow Trends @
<https://insights.stackoverflow.com/trends?tags=lisp%2Chaskell>
 Popularity of LISP dialects @
<http://blockml.awwapps.com/example/example/document.html?id=DIALECTS>
- [9] GitHub 2.0: A Small Place To Discover Languages In Github @
https://madnight.github.io/github//pull_requests/2020/2
- [10] Hacker News Front Page Trends @
https://toddschneider.com/dashboards/hacker-news-trends/?q=lisp%2Chaskell%2Cjava&f=title&s=text&m=items_count&t=year