FALSIFICACIÓN DE LA TEORÍA EVOLUTIVA CON MICROLISP

José del Carmen Rodríguez Santamaría
https://www.ejristos.com
Bogotá, Colombia

28 de febrero de 2022

Índice

1. INTRODUCCIÓN **2**

2. COMPUTADORES TIPO CUADERNO Y MICROLISP **4**
2.1. Un Computador tipo MicroLisp 0.1 5
2.2. Antecedents of MicroLisp 5
2.3. MicroLisp 0.1 sobre cadenas binarias 6
2.4. MicroLisp 0.1 sobre dígitos 7
2.5. MicroLisp 0.0 8
2.6. PG (Programación Genética) con MicroLisp 0.0 8
2.7. PG con MicroLisp 0.1 11

3. PG CON REGULACIÓN **12**
3.1. MicroLisp 0.2 15
3.2. PG con MicroLisp 0.2 15
3.3. MicroLisp 0.3 16
3.4. Loops, bucles o ciclos 16
3.5. Tolerancia de fallas 17
3.6. Editor avanzado de MicroLisp 0.3 19
3.7. PG con MicroLisp 0.3 21
3.8. El mar del sinsentido 22

4. EL COSTO DE LAS APROXIMACIONES **23**

5. CONCLUSIÓN **23**

Resumen

Una falsificación de la **Teoría Evolutiva (TE)** con **Programación Genética (PG)** sobre **Flipp** ya está disponible (Rodríguez, [13] 2021). Dado que Flipp es un lenguaje de programación extremadamente simple, uno naturalmente se pregunta: ¿Qué podría pasar cuando uno usa un lenguaje más poderoso? nuestra respuesta es **MicroLisp**, una familia de lenguajes de programación que extiende Flipp, y que admite bucles o ciclos. Éstos eventualmente producen códigos más cortos pero agregan normas sintácticas. El primer factor acelera la PG, pero el segundo la sobrecarga. El resultado neto es, según nuestras abstracciones y simulaciones, negativo: la PG en MicroLisp funciona más lento que en Flipp. De la misma manera, la **Evolución de la Perfección** también es obligatoria. Por otro lado, los bucles incluyen señales reguladoras, por lo que son útiles para modelar la regulación génica y por lo tanto, su estudio ilumina la pregunta: ¿los genes aceleran la evolución? Dado que los genes son entidades reguladas, nuestra respuesta es negativa y, por lo tanto, la falsificación de la TE se fortalece.

1. INTRODUCCIÓN

Cuando uno se mira a los ojos con un murciélago que cuelga de una viga, y observa su hermoso pelo diminuto junto con su rostro tierno, uno siente que él está cerca a nuestro corazón. La resolución materialista de este sentimiento es la **Teoría Evolutiva (TE)**: el murciélago y el hombre son hermanos porque tienen un ancestro común. Nuestra formulación de esa teoría es la siguiente:

Tenemos en primer lugar que las instrucciones del genoma para continuar con la vida son recetas verbales, es decir, instrucciones que están mediadas por un código, el llamado código genético, que puede redefinirse en millones de formas diferentes. En terminología moderna: **el Genoma es software**. De la misma manera, la **Evolución es un desarrollador de software** porque puede producir las instrucciones del genoma de la descendencia a partir de las de los padres y generalmente con algunas modificaciones. **La TE Verbal** ahora dice: la Evolución es el único desarrollador de software responsable del origen del murciélago y del hombre.

La TE Verbal es obviamente falsa. Esta es terminología humana para decir que dado la experiencia de cualquier ser humano mayor de 1 año, es inmediato rechazar la TE Verbal. En efecto, nuestra vida gira en torno a inventar, usar, probar, enseñar recetas. Un bebé debe inventar muchos procedimientos para expresarse. En lenguaje formal, **una receta es un algoritmo**. Cuando se transcribe un algoritmo a un lenguaje de programación, tenemos **software**. Entonces, los algoritmos son para los seres humanos más primitivos que el software.

El punto es que sabemos lo difícil que es la vida: componer algoritmos bien hechos es una tarea extremadamente compleja. Ésto significa que la ruta a el éxito está pavimentada en fracasos que son seguidos por más fracasos. Eso es tan crudo que la mayoría de los productos de cualquier naturaleza son de alguna manera incompletos o disfuncionales. Las empresas de Software lo ejemplifican a la perfección y por eso actualizan sus programas todos los días.

Para convertir la experiencia subjetiva de la complejidad en una predicción científica que involucre Evolución y Genética, debemos digerir nuestro diario sufrir:

Cuando algo no funciona, hacemos cambios e incorporamos conocimiento foráneo. En Genética, los cambios se denominan mutaciones y la incorporación de el saber hacer

se llama recombinación. Estas dos operaciones forman la **Evolución**. Hacemos mutación y recombinación en todo momento, por eso la Evolución es inmanente a nuestras vidas como lo es la **Evolución de la Perfección**: las soluciones mediocres evolucionan para ascender a una buena calidad para afrontar después una batalla por la excelencia. Esta es una observación universal y por lo tanto, equivale a una predicción: **La Evolución de la Perfección es obligatoria** en cada Proceso Evolucionario -aparte de casos simples que a nadie le importa. Así, para poner a prueba la Teoría Evolutiva, que aparecimos por Evolución, buscamos evidencias de la Evolución de la Perfección. Con ese fin, nos miramos a nosotros mismos: ¿estamos llenos de rasgos incompletos y disfuncionales? Aunque nosotros tenemos defectos, digamos, somos mortales, nuestra respuesta definitiva a esta pregunta es simple: No, no estamos llenos de órganos incompletos y disfuncionales, en cambio, somos perfectos. Si la Evolución fuera la explicación correcta de mi existencia, defectos terribles abundarían por todas partes en mi cuerpo, forma y función. Ésto es falso. Tal afirmación equivale a un rechazo de la TE.

Ahora bien, **nuestra responsabilidad ha sido mostrar generosamente cómo pasar de afirmaciones intuitivas a hechos de Ciencia**, es decir, cómo hacer pruebas naturales, objetivas y claras. Ahora bien, dado que la TE trata con software, el marco natural para probarla es la **Programación Genética (PG)**, el uso de la Evolución para desarrollar software. Nosotros implementamos este programa sobre Flipp, un lenguaje de programación para una computadora con una celda de memoria. Este juego de niños fue suficiente para falsificar la TE: nuestras simulaciones muestran que la Evolución sí funciona, pero esto sucede a un ritmo tan lento que surge una predicción: tanto nuestros cuerpos como el registro fósil deben llenarse de todo tipo de imperfecciones. O bien, la Evolución de la Perfección es inevitable. Como esto es claramente falso, la TE ha sido falsificada (Rodríguez, [13] l.c., 2021, este trabajo será referido a continuación como **PG1**).

La razón para el artículo presente es la siguiente:

Encontramos en nuestro estudio que la Evolución puede reproducir exactamente pequeñas cadenas binarias pero que es incapaz de adivinar las largas, cuyas aproximaciones navegan por la mediocridad. A partir de ésto, especulamos que las pequeñas cadenas adquirirán una función y evolucionarán hacia la perfección. En Biología Molecular, a los pequeños fragmentos de ADN con función se denominan genes. Por lo tanto, nuestra especulación dice: dado que Evolución no puede adivinar cadenas largas, su capacidad de hacer coincidir las cortas se traduce naturalmente en el surgimiento y proliferación de genes.

Consideramos en este momento que esta especulación no puede ser efectivamente falsificada, por lo que la aceptamos. Entonces, supongamos que los genes existen: ¿Los genes aceleran la Evolución? Con esto queremos decir que la introducción de genes reduce el costo de la perfección de ser exponencial en la longitud de la entrada a, digamos, polinómica o tal vez, logarítmica.

Antes de estudiar la cuestión antes mencionada, debemos hacer una precisión: los genes que conocemos no son fragmentos de ADN. Un gen es una cadena que codifica para un polipéptido y cuya síntesis puede ser regulada de acuerdo a las necesidades del momento. Por lo tanto, debemos fabricar un entorno evolutivo en el que la regulación esté disponible. Presentamos una solución a esta demanda de la siguiente forma: extendemos Flipp a una familia de lenguajes de programación que incluyen instruc-

ciones regulativas para estudiar más a fondo la batalla en PG de la Evolución contra la complejidad. Más explícitamente: fabricaremos MicroLisp, una familia de lenguajes de programación, algunos de los cuales incluyen bucles. Estos dicen cuántas veces una determinada tarea debe ser ejecutada. Por lo tanto, los bucles incluyen señales reguladoras que eventualmente permiten códigos más cortos. Esto acelerará la Evolución. Pero por el otro lado, los bucles sobrecargan a la Evolución con reglas sintácticas. Esto disminuye la velocidad de Evolución. ¿Cuál podría ser el resultado neto? Para producir una respuesta autorizada, haremos aquí algunas simulaciones. Entonces, comencemos con la construcción de MicroLisp.

2. COMPUTADORES TIPO CUADERNO Y MICRO-LISP

Los seres humanos utilizamos hojas de papel y lápiz en todas partes para hacer cálculos. Supongamos, por ejemplo, que necesitamos hacer la contabilidad del movimiento de nuestra tienda. En referencia a un cuaderno, un pseudo-código para esa tarea se ve así:

1. Asigne la primera página a las manzanas, la segunda a las naranjas y la tercera a las bananas.
2. Todas las páginas se inicializan con cero.
3. Llegó una manzana: ve a la primera página y suma 1.
4. Llegó una naranja: vaya a la segunda página y agregue 1.
5. Saca una manzana: ve a la primera página y resta 1.
6. Llegó un banano: vaya a la tercera página y agregue 1.
7. La tienda se cerró a las 6 pm. Se necesita un informe de la actividad en el día. Por lo tanto, imprime el número de manzanas, naranjas y bananos que hay en la estantería.
8. Fin.
9. Salida: 0 1 1

Tenemos aquí operaciones elementales que se pueden abstraer fácilmente mediante simples comandos: avanzar una página, retroceder una página, agregar uno a la página actual, restar uno de la página actual, imprimir el contenido de la página actual. Esta observación nos permite idear una computadora junto con su sistema operativo.

Nuestra computadora en mente es conceptualmente demasiado simple. Esto es maravilloso porque el ADN también es conceptualmente simple y no solo él sino el funcionamiento de todo el maquinaria bioquímica y genética. La complejidad surge de la gran cantidad de tipos de componentes que siempre están en el escenario.

2.1. Un Computador tipo MicroLisp 0.1

Presentemos una **Computadora tipo Cuaderno** y su idioma de programación **MicroLisp**. La computadora tiene una matriz de memoria con 5 celdas en fila y un puntero dirigido a la única celda activa. Una celda debe estar seleccionada, activa, en cada instante. Para navegar circularmente por la matriz, utilice el comando $<$ para seleccionar una celda a la izquierda, $>$ para seleccionar una celda a la derecha.

Las celdas de memoria guardan números de un conjunto dado. Trabajaremos con dos alfabetos de números. Primero: números binarios 0, 1. Segundo: dígitos 0, 1, 2,... .9. Estos conjuntos se denominan **alfabetos de celdas** y son iguales para todas las celdas. La matriz de memoria es circular, así como el alfabeto para las celdas de memoria. Al inicio, todas las celdas de la Computadora tipo Cuaderno mantienen un cero.

Desafío. Haga una computadora como se ha especifico usando una serie de ruedas, que tienen un símbolo en cada diente. Una rueda activa gira alrededor paso a paso de acuerdo con los comandos de MicroLisp. El puntero está en la parte superior y activa la celda que también está en la parte superior de la rueda.

MicroLisp 0.1 tiene 5 comandos (imaginamos aquí que la matriz de memoria está inmóvil, como en las computadoras de estado sólido, mientras que el puntero se mueve por la matriz):

$+$ Agrega uno a la celda actual.

$-$ Resta uno de la celda actual.

$>$ Mueva el puntero una celda a la derecha.

$<$ Mueva el puntero una celda a la izquierda.

. Lea el valor de la celda activa e imprímalo.

2.2. Antecedents of MicroLisp

El nombre **MicroLisp** es una concatenación de Micro y Lisp. Micro significa que nuestra computadora es elemental. El lenguaje de programación Lisp fue inventado por John McCarthy, alrededor de 1955 y se deriva de "Procesador de listas"(Two-Bit History [15] 2018). Ese nombre se usa aquí porque cada programa MicroLisp es una lista de comandos de programación. Por otro lado, Lisp es un poderoso lenguaje de programación de alto nivel, cuya influencia se extiende por todo el mundo y su nombre es un desafío para nosotros: ¿podemos hacer de MicroLisp algo tan poderoso y encantador como Lisp?

MicroLisp 0.1 es una abstracción simplista del procedimiento de contabilidad que todo el mundo hace en un cuaderno. Su simplicidad se explica porque es todo lo que necesitamos para hacer PG que pueda reproducir las contraseñas binarias. Para comprender la importancia de nuestra abstracción, necesitamos un contexto. El apropiado es el de Ciencias de la Computación y sus diversas abstracciones de computadoras, entre las cuales las siguientes han sido altamente influyentes para nuestro trabajo:

1. Las máquinas de Turing son computadoras todopoderosas. Abstraen el procedimiento: bajo esta situación, haz esto. Por lo tanto, tienen ramificación y toma

de decisiones como una propiedad incorporada. Fueron inventadas por Alan Turing en 1936 y se consideran el evento inicial de la ciencia de la computación (BitMerge [5] 2016, Davey [8] 2010).

2. P" fue una variación académica y mística de las máquinas de Turing creada por Corrado Böhm en 1964. Convirtió la computación en un juego para niños (McCaffrey [11] 2018). Seguimos ese trabajo, en compañía de muchos otros, dos de los cuales siguen a continuación.

3. Intercal fue creado solo por diversión por Don Woods y James M. Lyon en 1972, un lenguaje misterioso pero influyente (Bell [4] 2021).

4. Brainfuck es una variación de P", fue creado en 1993 por Urban Müller. Pone la sabiduría de Böhm en términos simples, sin su misticismo. El resultado es que a todos les encanta. Precisamente, el trabajo importante para nosotros fue su implementación en JS + HTML + CSS debido a Michael Duerinckx ([9] 2014). La idea de usar este lenguaje para PG nos vino de Kory Becker y Justin Gottschlich ([3] 2017). Ese lenguaje ha dado lugar a muchos dialectos, y nuestro MicroLisp es solo una deconstrucción creativa que persigue el sentido común y la PG. No obstante, consideramos que la abstracción original realizada por Böhm se presta a una comparación directa con trabajos matemáticos, que son abundantes.

2.3. MicroLisp 0.1 sobre cadenas binarias

Listemos algunos programas en MicroLisp 0.1 y sus outputs:

PROGRAMA	OUTPUT
.	0
+.	1
>+.	1
+.−.	10
+.>.	10
+..−.	110
+.>+.>.	110
+...−..	11100
+.>+.>+.>.>.	11100
+.−.+.−.+.−.	101010
>+.><−.>>>+.><−.>+.><−.	101010

Al mirar esta tabla, notamos tres hechos:

1. Hay muchos programas equivalentes que producen el mismo resultado. Por ejemplo, se pueden insertar bucles vacíos +− o <> en cualquier sitio sin provocar una modificación del comportamiento.

2. No hay un límite superior en la longitud de los programas cuya salida coincide con una cadena objetivo. Basta con insertar bucles vacíos para producir programas cada vez más largos.

3. Para cada cadena binaria, existe un programa MicroLisp 0.1 que la genera. Más concretamente, se pueden encontrar fácilmente programas con aproximadamente el doble de longitud que la cadena binaria de destino.

Estos resultados influirán en nuestro trabajo. Veáse más abajo.

2.4. MicroLisp 0.1 sobre dígitos

Los siguientes son ejemplos y ejercicios de programas separados e independientes cuando el alfabeto celular es el conjunto de dígitos:

- . lee el contenido de la celda c0 y lo imprime. Output 0.
- $+$. agrega uno a la celda c0 e imprime su contenido. Salida 1.
- $++$. suma 1 + 1 a la celda c0 e imprime su contenido. Salida 2.
- $+++$. Salida 3.
- $++++$. Output 4.
- 5?
- $+$.. Salida 11.
- 111?
- $-.>--.-$. Salida 987.
- Nos falta el símbolo de espacio en blanco. Entonces, podemos generar 11 también como 1 desde la celda c0 seguido de un 1 de la celda c1: $+.>+$.

Vemos que hay muchas formas como una secuencia puede ser cazada por MicroLisp 0.1. Además, cada cadena formada con los símbolos de nuestro alfabeto de comandos es un programa viable. Pero algunos programas, como $++++$, no producen ningún resultado.

1 ***Ejercicio.*** *Practique MicroLisp 0.1 con el programa* `MicroLispComputer01.html`*. Realice una prueba sobre cadenas binarias y luego sobre los dígitos. Acompáñese con niños y adolescentes. Ellos lo encontrarán genial. Nuestro material es la introducción mecanicista más comprensible a las computadoras y a su teoría.*

2 ***Ejercicio.*** *Nuestra Computadora tipo Cuaderno implementa un intérprete, es decir, un sistema que ejecuta cada comando tan pronto como se escribe. Implemente un compilador, un sistema que solo procesa programas completos. Nuestra respuesta se encuentra más abajo.*

Desafío. ¿Las células de los seres vivos tienen un intérprete de ADN o un compilador?

Nota para puristas. Podemos trabajar con secuencias decimales o binarias, así que tenemos un problema porque el programa

$+>+++.>.-.<.$

genera 3093 para secuencias decimales, pero 1011 para binario. Así, en sentido estricto, tenemos dos idiomas diferentes. Si prefiere ser estricto, utilice *R* y *L* para secuencias decimales, pero $<$ y $>$ para binarias. Con este convención, obtenemos:

El programa $+R+++.R.-.L.$ genera 3093, mientras que el programa $+>++$ $+.>.-.<.$ da como resultado 1011. Esto puede verse bien, pero a continuación posiblemente descubriremos que esta directiva sobrecarga nuestra cabeza y que asumir que el sentido sea dependiente del contexto es más adecuado para los seres humanos.

Establezcamos ahora la relación de Flipp con MicroLisp 0.1.

2.5. MicroLisp 0.0

Hemos prometido extender Flipp.

3 ***Ejercicio.*** *Demuestre que MicroLisp 0.1 extiende Flipp (Rodríguez, [13] l.c., 2021). Sugerencia: trabaje con un subconjunto de MicroLisp 0.1 y use solo una celda de memoria. El lenguaje resultante se llama* ***MicroLisp 0.0****. Hay al menos dos versiones, y son bastante similares. Respuesta*

Los siguientes son algunos ejemplos de programas en **MicroLisp 0.0** y sus resultados:

PROGRAMA Y SALIDA	
.	0
.....	00000
+.	1
+++.	1
+.+.	10
+.+++.	10
+..+.	110
+..+++.	110
+...+..	11100

‘

Esta tabla ilustra cómo el programa $++$ no hace nada y cómo crea variabilidad.

2.6. PG (Programación Genética) con MicroLisp 0.0

Flipp y MicroLisp 0.0 son muy similares. ¿Son iguales de rápidos?

Nuestra tarea es decidir si la PG sobre MicroLisp 0.0 es más rápida que sobre Flipp para cazar cadenas binarias aleatorias. Nuestra solución es la siguiente:

Tanto MicroLisp 0.0 como Flipp tienen dos símbolos que representan comandos, por lo que, en este respecto, están en pie de igualdad. Sin embargo, para simular Flipp sobre MicroLisp, uno necesita en general programas más largos. Esto debe aumentar el costo de la PG. Pero para cada cadena, hay uno y sólo un programa Flipp que la caza, mientras que MicroLisp admite muchas variantes. Esto podría disminuir el costo de PG. ¿Cuál podría ser el resultado general?

Nuestro plan es usar la Evolución para adivinar un Programa MicroLisp para reproducir una cadena binaria objetivo. Nuestra abstracción no está tan lejos de la biología: estamos usando la Evolución para adivinar un tipo de ADN que es un algoritmo para sintetizar proteínas cuyo contenido está codificado como secuencias binarias.

Nuestra PG (Programación Genética) se define de la siguiente manera:

- Declaramos una cadena binaria, digamos, 100110, como la cadena objetivo.
- Le damos a Evolución la tarea de sintetizar un programa MicroLisp 0.0, algo como $+.-..+..$, que debe generar la cadena o contraseña objetivo.
- Una vez que la cadena de destino ha sido resuelta, se da una más larga, digamos 1110011.
- Utilizamos gradualismo, en el que las contraseñas se adivinan paso a paso, como en un juego de niños. Por tanto, no podemos decir nada sobre contraseñas de seguridad como las utilizadas en los cajeros automáticos, pues son todo o nada.

Para implementar este plan, necesitamos componer un compilador para MicroLisp 0.0. Hasta ahora tenemos un intérprete para MicroLisp 0.1, un programa que ejecuta un comando tan pronto como llega. Pero un compilador ejecuta programas, una lista de comandos

Para ello, decidimos adaptar el programa `GPwithFlipp.html` de PG1 a MicroLisp 0.0. Buscando sencillez, se procedió de la siguiente manera: Diseñamos una función en JS que produce automáticamente el programa en MicroLisp 0.0 más corto que caza una cadena dada. Por ejemplo, la cadena binaria 10010 coincide con el output del programa $+.+..+.+.$ que tiene una longitud de 9. De manera similar, la cadena 11111 es la salida de $+.....$ que tiene una longitud de 6. Así que la longitud del programa desarrollado automáticamente se midió y se le dio a la PG la tarea de encontrar ese programa entre todos los programas de precisamente esa longitud. Esto fue hecho por el programa `GPwithMicroLisp00.html`. El programa arrojó que el costo para PG fue $51{,}8e^{0{,}34x}$, para aleatoriedad $0{,}6e^{1{,}04x}$, para búsqueda en el peor de los casos $1{,}3e^{1x}$. Todos los valores fueron más altos que para Flipp, que para PG fue de $49,7e^{0{,}3x}$ según PG1. La razón es que las cadenas con una mezcla de ceros y unos necesitan programas más largos y, por lo tanto, son más difíciles de encontrar, ya sea por aleatoriedad o por Evolución. Sin embargo, la PG es, con mucho, la mejor en comparación con la aleatoriedad y la búsqueda en el peor de los casos.

La solución anterior no es muy honesta porque incluye una pista, la longitud de la solución. ¿Qué podría pasar si uno no da ninguna pista? Al atacar esta pregunta, nos encontramos con varios problemas:

1. Si la cadena binaria tiene una longitud n y tiene ceros y unos, ningún programa de longitud n es bueno para cazarla. Por ejemplo, el programa para generar la cadena 1010 de longitud 4 es $+.+.+.+.$ y tiene de largo 8 y este es el más corto. Pero de otra parte, la cadena 1111, de la misma longitud, es cazada por el programa $+....$ que tiene una longitud 5. Entonces, cuando la Evolución no sabe la longitud del programa solución, la Evolución debe tener acceso a diferentes longitudes.
2. Muchos programas MicroLisp 0.0 pueden cazar la misma cadena binaria pero pueden tener diferentes longitudes. Por ejemplo, la inserción de $++$ no produce cambio de la salida pero cambia la longitud de su programa.
3. Programas como $+++++$ no tienen output.

Nuestra solución fue cambiar la implementación de la Evolución. En lugar de trabajar con cromosomas de la misma longitud, permitimos que la longitud varíe entre x y $2{,}5x$, para una cadena binaria de longitud x. La recombinación se redefinió para aceptar recombinantes de diferentes longitudes, y la mutación se implementó con tres opciones: puntual, cambio de un símbolo por otro, e inserción o eliminación de un símbolo en un sitio dado. También redefinimos el puntaje o aptitud de un programa dado: además de una recompensa por lograr una coincidencia en un sitio, agregamos una bonificación para la aproximación de la longitud. La razón es que un programa puede generar una cadena de 15 caracteres de largo para que coincida con una cadena de longitud 8. Esa cola innecesaria merece un castigo.

Oficialmente, la idoneidad de un programa se define como la suma del número de coincidencias más una bonificación por aproximación a la longitud. Ejemplo:

```
//Se calcula la aptitud de un programa MicroLisp en gradualismo
//por comparación de su salida con la contraseña de destino:
//aptitud =  coincidencias + bonificación por coincidencia de longitud.
//Ejemplo:
//contraseña = 1010
//salida =     1111000
//coincidencias = 2 (1010 y 1111 tienen dos caracteres en común).
//Castigo por diferencia de longitud = 7-4 = 3.
//Bono máximo por coincidencia de longitud = longitud de contraseña = 4
//Bono real = Bono máximo - castigo = 4 - 3 = 1
//Puntaje = coincidencias + bono real = 2 + 1 = 3.
```

A menos que se indique lo contrario, utilizaremos esta aptitud o puntaje a continuación.

Todo esto se incorporó al programa `GPwithMicroLisp00v2.html`. El resultado neto es que la PG sobre MicroLisp 0.0 v2 (sin pistas) se ejecuta en el régimen exponencial: una corrida en el intervalo [1, 18) generó un costo para la PG de $25{,}7e^{0{,}67x}$, para la aleatoriedad de $2{,}7e^{0{,}78x}$ y en el peor de los casos de búsqueda $1{,}2e^{0{,}99x}$. Estos valores son mucho más altos que para Flipp. Sin embargo, recordemos los costos de MicroLisp 0.0, que tenía la pista de la longitud: para la PG $51{,}8e^{0{,}34x}$, para el azar $0{,}6e^{1{,}04x}$, para el peor de los casos de búsqueda $1{,}3e^{1x}$. Esto demuestra que la pista de la longitud es muy efectiva pero es incapaz de destronar al régimen exponencial.

Nota matemática. El costo de la PG sobre Flipp fue de $49{,}7e^{0{,}3x}$ según GP1. Esto contrasta con el costo encontrado aquí para la PG sobre MicroLisp 0.0 sin pistas, que fue de $25{,}7e^{0{,}67x}$. Para comparar extrapolaciones, prestamos atención a los exponentes. En el primer caso fue $0{,}3x$ y en el segundo $0{,}67x$ que lo aproximamos a $0{,}6x$. La diferencia no es tan grande. Sin embargo, esta pequeña diferencia provoca enormes desproporciones en el costo, la expresión exponencial completa. Notemos que $(e^{0{,}3x})^2 = e^{0{,}6x}$. Esto quiere decir que para $x = 47$, el primer costo en individuos es $e^{0{,}3\times 47} = 10000000$ mientras que el segundo debe ser su cuadrado, 100000000000000. Por eso pudimos ejecutar rápidamente simulaciones sobre Flipp, pero aquí uno debe armarse de mucha paciencia. Y más aún para las próximas versiones de PG.

En general, la mayoría de las diferencias encontradas por nosotros se encuentran en el ámbito cuadrático y rara vez en el cúbico. Sin embargo, hay una excepción importante: el Darwinismo en su valor nominal es una pista tan poderosa que el costo de un algoritmo genético que coincide con cadenas binarias es logarítmico en lugar de exponencial. Su coste es incluso más bajo que uno polinómico. Eso es magia pura: es muy interesante entender claramente de dónde viene (Rodríguez, [14], 2021).

4 ***Ejercicio.*** *Compare el costo* $e^{0,3x}$ *con* $e^{0,31x}$. *Respuesta*

5 ***Ejercicio.*** *Siempre afirmamos que el costo de la PG es exponencial. Resuelva la objeción de que el ajuste natural de cada conjunto finito de datos es un polinomio y no una función exponencial. Respuesta*

Desafío. Estamos acostumbrados a imaginar que podemos discernir por inspección cuando una diferencia tiene significación estadística. Sin embargo, no tenemos una base sólida para afirmar que las diferencias encontradas en este trabajo tengan algún valor predictivo. ¿Resolvería esta deficiencia? Nunca olvide que lo importante son las extrapolaciones de valores desde rangos menores a 100 hasta el reino de los miles de millones.

Pasemos ahora a evaluar el costo de la PG con el conjunto completo de comandos de MicroLisp.

2.7. PG con MicroLisp 0.1

Nuestro comportamiento habitual con los cuadernos para hacer procedimientos contables se ha resumido en MicroLisp 0.1: navegar por una matriz de memoria, navegar por el alfabeto de celdas, e imprimir el contenido de una celda. Tenemos la opción de trabajar con dígitos además de números binarios. Sin embargo, aquí preferimos las secuencias binarias porque nos permiten ver las líneas de regresión más rápido. Estas líneas son la herramienta que posiblemente nos permita extrapolar los resultados de nuestras muy humildes simulaciones al mundo inalcanzable de los grandes números, el habitual en Biología Molecular.

6 ***Ejercicio.*** *Decida si la PG de cadenas binarias aleatorias sobre MicroLisp 0.1 es más rápida que sobre Flipp. La pregunta es interesante porque de un lado, MicroLisp permite más opciones para construir la misma cadena entonces, podría ser más rápida. Pero por otro lado, el alfabeto de MicroLisp es más largo, por lo que podría ser más lenta. Decida cuál es el resultado neto. Respuesta*

Tenemos la pretensión de que los costos reportados por nuestras líneas de regresión predicen la respuesta a cadenas largas como los que requiere la Biología Molecular. El problema es que las cosas no son tan simples. En ese sentido, el siguiente ejercicio amplía nuestra visión.

7 ***Ejercicio.*** *El programa* `FlippDynamics.html` *de PG1 muestra cómo la PG intenta encontrar programas en Flipp que cazan cadenas binarias asociadas con tableros de ajedrez de tamaño predefinido. Adapte ese programa a MicroLisp 0.1 y explore*

su ejecución para tamaño creciente hasta tableros de lado con 256 celdas. Describa lo que sucede. Proponga una explicación y póngala a prueba. Respuesta

Desafío. El programa `MLisp01GPDynamics.html` presentó a la Evolución como una fuerza de estancamiento que frustra el Proceso Evolutivo. ¿Podremos extrapolar este resultado negativo a longitudes largas para proclamar que la Evolución es incapaz de lograr la perfección bajo alta complejidad? Nuestra respuesta es la siguiente: el programa `MLisp01GPDynamics.html` muestra que la Evolución se estanca, pero que tiene fluctuaciones. Consideramos que éstas nunca mueren porque vienen de la aleatoriedad y que gracias a ellas, la Evolución puede producir perfección si tan solo tiene suficientes recursos, digamos, tiempo infinito. Así, aceptamos las extrapolaciones dadas por las funciones exponenciales que son generadas por el programa `GPwithMicroLisp01.html`. Ponga a prueba nuestra creencia: si las fluctuaciones son la clave para llegar a la perfección, entonces pueden potenciarse aumentando los efectos de muestreo. Para hacer esto, disminuya el tamaño de la población a 2, y pruebe si el el proceso evolutivo hacia la perfección es más rápido.

En general, nuestro credo es el siguiente: la Evolución tiene dos componentes, uno que batalla por el estancamiento y el otro por la libertad. El primero es determinista, el segundo al azar. Imaginamos que la creatividad de la aleatoriedad no es silenciada por los efectos deterministas. Ahora bien, el hecho de que la Evolución sea mejor que la aleatoriedad para la síntesis de software nos dice que las sucesiones de pequeñas mutaciones con efectos graduales juegan un papel pequeño pero observable y que crea un impulso determinista en la Evolución. Entonces, la PG pone al determinismo a trabajar sobre las propuestas de la aleatoriedad.

8 ***Ejercicio.*** *Determine el costo de la PG sobre la versión de MicroLisp cuyos comandos son:* $< +- >$ *.... que hacen que la probabilidad de imprimir sea igual a 0,5. Este alfabeto ha demostrado ser bueno para luchar contra la complejidad.* Respuesta

3. PG CON REGULACIÓN

Cada mutación en un programa en MicroLisp 0.1 es viable. La razón es que MicroLisp 0.1 fue diseñado para ser demasiado simple. Pero contrario a esto, la mayoría de las mutaciones son deletéreas o letales en Biología. Entonces, tenemos una pregunta: ¿qué extensión única debe considerarse para proporcionar a MicroLisp más posibilidades y también más fragilidad para que algunas mutaciones puedan causar la muerte? Nuestra respuesta: incluir la regulación es la única extensión de MicroLisp 0.1 que genera la máxima potencia y fragilidad al mismo tiempo. Nuestra elección hace justicia a la Biología porque la regulación es omnipresente en la Vida, incluso a nivel de bacterias. Su descubrimiento les valió un Premio Nobel para Jacob, Lwoff y Monod ([12] 1965), quienes demostraron que los genes pueden activarse y desactivarse al cambiar el alimento que se les da a las bacterias.

Definición. Para cada lenguaje de programación, incluido el ADN, una instrucción regulativa es aquella que se refiere a otras instrucciones o a una propiedad de diseño.

El ADN es un lenguaje de alto nivel que admite regulación (Bodine [6] 2014, Virtual Genetics Education Center [16] 2015).

En el lenguaje del ADN, los efectos reguladores multiplican la velocidad promedio de expresión por un número real que puede variar de 0 a 10 o más (multiplicar por 0 significa apagar, multiplicar por 1 significa encender, multiplicar por 0,5 es una inhibición, multiplicado por 2 es una mejora). La regulación génica asegura que los genes apropiados se expresen en los momentos y lugares apropiados. La regulación génica también puede ayudar a un organismo a responder de manera óptima a su entorno.

La regulación es una pesadilla en la ingeniería. Una forma muy práctica de aprender esto es intentar arreglar en una bicicleta un problema con sus cambios, desviadores, cadena, piñones, o de su posición en relación al tubo del sillín. O piense en la tensión de los cables de los frenos o de los cambios. Demasiada tensión y la bicicleta sufrirá daños perjudiciales o letales, pero ante una tensión baja y la función desaparecerá con consecuencias posiblemente peligrosas. Todo esto es tan difícil que con mucho gusto le pago a un técnico. Pero me parece muy instructivo hacer al menos un intento hasta el éxito.

Miremos un ejemplo de biología.

Considere las plantas con hojas. En el envés sobre su epidermis, las hojas tienen estomas, que son poros que se abren al medio externo. Pero estos poros no son estáticos. Cada estoma tiene una geometría variable, y gracias a ello puede estar abierto o cerrado en diversos grados. Haciendo uso de esta libertad, las hojas regulan la velocidad de intercambio de la materia con el medio exterior. Los estomas son las puertas para absorber el dióxido de carbono y para expulsar el oxígeno y el agua que fue absorbida por las raíces. El agua es expulsada en forma de vapor.

Los estomas dan lugar a muchas consideraciones con preguntas específicas de ingeniería o arquitectura funcional:

- ¿Qué pasa si no hay ningún estoma en absoluto?
- ¿El intercambio es pasivo o activo?
- ¿Es el transporte un problema global, indivisible, o depende de la sustancia?
- ¿Los poros se comunican con el medio desnudo intercelular o con una membrana de células especializadas?
- Dado que el metabolismo de cada planta está dirigido a absorber óxido de carbono y expulsar oxígeno, ¿estará ligado el transporte de estos dos componentes? En caso afirmativo, el diseño se sobrecargará, pero la eficiencia energética aumentará.
- ¿Qué pasa con el tamaño y la forma que garanticen la máxima apertura y el cierre total?
- ¿Qué pasa con la densidad de estomas? Tenga en cuenta que, hasta cierto punto, el tamaño y la densidad son independientes porque los estomas pueden ser pequeños y estar separados.
- ¿Qué pasa con el material para hacer celdas con geometría variable?

- ¿Qué pasa con el grosor de las paredes?
- ¿Cuántas paredes debe cubrir una celda de estoma? ¿Dividiríamos las funciones de un pared en uno o dos roles principales y asignaríamos cada rol a una pared diferente? Por ejemplo, se puede trabajar con una pared y poner todo allí. O podríamos considerar que la protección y el resto de funciones son dos objetivos tan inmiscibles que merecen dos paredes separadas. ¿O trataríamos de explicar por qué las células vegetales tienen tres paredes más otra para separar las células? (Bailey, [2] 2019).
- ¿Qué pasa con la estabilidad de su diseño bajo diversas tensiones extremas, por ejemplo, presión de dióxido de carbono demasiado baja o sequía prolongada? Notemos que tenemos plantas que viven dentro del agua, como las algas, y otras que pueden absorber el agua de las nubes, como algunas en los altos Andes del Perú.

En nuestra opinión, la mayoría de estas propiedades evolucionan gradualmente, al menos dentro de ciertos rangos. Pero si uno se acuerda de la bicicleta, se espera que los rangos sean muy estrechos. Por ejemplo, una geometría variable depende de la elasticidad de las capas de las paredes más la metaestabilidad, la propiedad de adoptar una forma completamente diferente debido a un pequeño cambio. Demasiado baja elasticidad y la respuesta podría estar ausente o demasiado tarde, pero si fuese demasiada se necesitaría un ajuste fino de los sensores de tensión superficial del plasma celular. Así, nuestra discusión predice una Evolución gradual múltiple que estaría bordeada por todos lados por una pérdida de función, es decir, por la muerte.

Otro ejemplo de Biología: la regulación génica de la concentración de ARNm.

Un cromosoma es como un libro con muchos capítulos. Un gen es como un capítulo. No está permitido llevarse a casa ningún volumen. En su lugar, se le permite hacer una fotocopia de un capítulo. Ésta es la transcripción de ADN a ARNm con la m de mensajero. Con su fotocopia, vaya a un ribosoma y solicite una traducción de su ARN a una cadena polipéptída. Si su fotocopia no desaparece, los ribosomas continuarían reutilizándola para hacer más cadenas. ¿Hasta cuándo? Hasta agotar uno de los aminoácidos utilizados. En ese momento, toda la maquinaria de la vida se detiene porque cada todo aminoácido es esencial. Por lo tanto, si no se destruye una sola molécula de ARNm, se crea como mínimo una mutación muy perjudicial. En el otro extremo, su ARNm puede desaparecer tan rápidamente que nunca llegue a un ribosoma, o el ribosoma podría ser demasiado lento para prepararse.

Por lo tanto, el problema es regular la descomposición del ARNm para proporcionar el sistema más estable y eficiente. La solución adoptada en la Naturaleza es disfrutar de la larga vida del ARNm y provocar una degradación inducida con un conjunto de enzimas: cada molécula de ARNm está marcada con una señal que transmite el mensaje de la rapidez con la que debe degradarse. La velocidad de degradación debe ajustarse finamente a las necesidades del organismo. Por lo tanto, una enzima lee y ejecuta el mensaje al preparar el ARNm para una rápida degradación por exonucleasas o exosomas (Hoof y Parker [10] 2002).

Desafío. Discutir un caso de regulación génica que es históricamente muy importante: los pinzones de Darwin (Arhat [1] 2009, Zandonella [17] 2015). ¿Considera Usted que este caso de Evolución verificada convierte a la TE en una verdad científica?

Volvamos ahora a MicroLisp. Tenemos la ocasión de usar la regulación para solucionar una frustración con MicroLisp 0.1: no podemos separar los números con un espacio. Por esa razón, no tenemos idea del significado de la cadena 1158. ¿Es 1 1 5 8? ¿O mejor 1 15 8? O solo 1158? Implementemos una solución a este problema y midamos el costo de hacer PG en el escenario subsiguiente.

3.1. MicroLisp 0.2

Para imprimir números separados, debemos incluir un espacio en blanco como parte del alfabeto del idioma. Estamos lidiando con un problema regulatorio porque pedimos imprimir en blanco después de cada número. Nuestra solución es concatenar el símbolo de blanco a cada número para luego imprimir la cadena resultante. La forma más fácil de hacer que el espacio en blanco forme parte del lenguaje es declarar el alfabeto celular como los dígitos unidos al símbolo en blanco. La nueva variante se llama MicroLisp 0.2. Entonces, el nuevo alfabeto de celda es 0123456789*b*, con un espacio en blanco en el último sitio que ha sido reemplazado por *b*. Una celda de memoria puede mantener uno y solo un símbolo de esta lista. En este nuevo alfabeto, la suma y la resta no están definidas en todas partes. Por lo tanto, la navegación a lo largo del alfabeto de celdas debe adaptarse un poco. Veamos cómo:

MicroLisp 0.2 tiene 5 comandos:

\+ avanzar un lugar en el alfabeto de celdas.

\- moverse un lugar hacia atrás en el alfabeto de celdas.

$>$ Mueva el puntero de activación una celda hacia adelante.

$<$ Mueva el puntero una celda hacia atrás.

. Imprima el valor de la celda activa.

MicroLisp 0.1 y 0.2 se relacionan así:

Agregar uno en la versión antigua es una operación que coincide en la nueva con avanzar un lugar en el alfabeto para los números 01...8. Pero los dos comandos difieren en cuanto a sumar uno a 9. En la versión anterior, 9 + 1 da como resultado 0, pero ahora devuelve un espacio en blanco. Por lo tanto, tenemos dos dialectos de MicroLisp. En general, para cada decisión que tomamos, hay muchas variantes posibles que pueden crear diferentes dialectos. Entonces, estamos tratando con una gran familia de lenguajes de programación de computación, pero estudiaremos solo algunos exponentes. La forma de comparar su desempeño sobre cadenas binarias es completamente análoga.

9 ***Ejercicio.*** *Practique MicroLisp 0.2 con el programa* `MicroLispComputer02.html`.

3.2. PG con MicroLisp 0.2

Midamos ahora el costo de introducir la regulación en MicroLisp.

10 ***Ejercicio.*** *Compare el costo de adivinar una contraseña binaria aleatoria con el de adivinar la misma cadena a la que se ha adjuntado b (de espacio en blanco) al final. Este ejercicio nos enseña lo costoso que es introducir el elemento de regulación más simple en PG.* *Respuesta*

3.3. MicroLisp 0.3

Nuestros lenguajes MicroLisp 0.0, 0.1 y 0.2 son tolerantes a las mutaciones: todas las mutaciones son viables y los compiladores correspondientes las pueden entender sin ambigüedades. Esto es diferente a la biología, en la que casi todas las mutaciones son perjudiciales o letales. Esto explica el siguiente llamado:

Concurso. Encuentre la extensión más simple para MicroLisp que mejore su expresividad, pero que también cree suficientes problemas sintácticos como para que se generen mutaciones letales en gran proporción.

Nuestra respuesta a este desafío es introducir bucles: pueden producir códigos más cortos, por lo que no se excluye que el costo promedio de la PG disminuya. Pero, como se exige, la sintaxis se sobrecargará lo suficiente como para permitir naturalmente la simulación de mutaciones letales. De hecho, el lenguaje contendrá un conjunto más grande de comandos y esto hace que el costo se dispare porque se necesitan más combinaciones para probar. Y, por otro lado, los bucles implican correlaciones entre símbolos diversos que pueden estar ampliamente separados. Por ejemplo, ¿qué haremos con un símbolo que ordene cerrar un ciclo cuando no se ha abierto ninguno? ¿Lo ignoramos? ¿Supondremos que hay un símbolo de apertura de ciclo 3 caracteres hacia atrás? ¿Debemos castigar la cadena con letalidad? Así, la sintaxis de los loops produce una gran cantidad de programas que no pueden ser interpretados de una sola forma. Nuestra decisión es utilizarlos como oportunidades para introducir mutaciones letales. Este es el propósito de los bucles en MicroLisp 0.3.

3.4. Loops, bucles o ciclos

Las llamadas repetidas a la misma porción de código son comunes en todas partes en la programación.

Ejemplo. El siguiente pseudocódigo repite la misma tarea 5 veces

```
tarea
tarea
tarea
tarea
tarea
```

Este código puede resumirse en el siguiente:

```
Repita la tarea 5 veces
{ tarea }
```

La primera instrucción es regulativa porque se refiere a la segunda.

¿Cómo se ejecuta este bucle o repetidor? Cuando la tarea se realiza por primera vez, el 5 se cambia a 4, luego a 3, 2, 1, 0 y luego se sale del ciclo. Del mismo modo, el código correspondiente en MicroLisp tiene el siguiente aspecto:

```
5 (contador de bucle)
[tarea]
```

y se ejecuta de la misma manera. Formalmente:

```
MicroLisp 0.3 tiene 7 comandos,
dos se refieren a la navegación a través del alfabeto celular,
dos a la navegación sobre la matriz de celdas de memoria,
dos son controles de bucle,
uno es el comando de impresión:
+ avanzar un sitio en el alfabeto en la celda actual.
- retroceder un sitio en el alfabeto en la celda actual.
> Mueva el puntero una celda hacia adelante en la matriz.
< Mueva el puntero una celda hacia atrás en la matriz.
[comienzo del bucle.
] final del bucle.
. Imprime  el valor de la celda activa.
```

Ejemplo. El programa

```
+++
[>+.<-]
```

pone el contador de bucle en 3 en la celda C0, $>+$. es la tarea a repetir que afecta a la celda C1. La subrutina $<-$ devuelve el puntero a la celda C1 y disminuye el contador. Se sale del bucle cuando el contador es igual a 0. Cuando el alfabeto de la celda son los dígitos y con y 8 celdas de memoria, el programa genera 123. Con más detalle: el contador se establece en tres en la celda C0. A continuación y dentro del bucle, la celda C1 adquiere el valor 1 que se imprime a continuación, y el contador se decrementa a 2. La segunda vuelta decrementa el contador a 1, y se incrementa en 1 el valor de C1 que adquiere el valor 2, que es impreso. La tercera ronda decrementa el contador a 0, e incrementa 1 al valor de C1 que ahora mantiene un 3, que se imprime. Dado que el contador vale 0, se sale del bucle. Salida: 123.

3.5. Tolerancia de fallas

Para embarcarnos en PG 0.3, que admite bucles, debemos tener en cuenta que encontraremos programas mal elaborados. Ejemplos:

```
+++[>++++.      Solitario [
+++]>++++.      Solitario ]
+++]>++.[       Par invertido ][
>[>+++].        Un bucle vacío
+++[++.]        Un repetidor infinito
```

La pregunta es: ¿Cuán tolerante a fallas debe ser nuestro lenguaje? Hasta ahora, los únicos programas mal inventados eran aquellos sin salida, como $+>>>+++$, y estos se ejecutaban sin ninguna prueba de verificación. Entonces, hemos sido tolerantes a fallas y esto podría ser bueno para la Evolución: una sola mutación que incorpore un comando de impresión es suficiente para producir una salida. Otra forma en la que hemos introducido la tolerancia a fallas es evitando el desbordamiento de la memoria. Con ese objetivo, hemos declarado que la matriz de memoria es circular. Esta también

es una medida que elimina la complejidad porque los símbolos específicos se pueden encontrar de dos maneras diferentes.

Ahora bien, en general y en nuestra opinión, la biología da la bienvenida a todas las opciones posibles con respecto a la tolerancia a fallas. Para ver esto, solo considere que los animales no se discriminan a sí mismos por la perfección física, pero pueden hacerlo por reconocimiento químico y por comportamiento.

Ejemplo. Un bebé perfecto y corpulento nació de una pareja de lobos. Tenía una tendencia extrema a ser dominante y desobediente. Fue tan problemático que el padre lo mató. Ahora, en retrospectiva, nos preguntamos: ¿Fue la naturaleza tolerante a fallas? Nuestra respuesta es negativa. Para los lobos que necesitan ser disciplinados y muy cooperativos, esa fue una decisión muy sabia porque un lobo problemático podría llevar a la manada a la extinción.

Ejemplo. La inmensa mayoría de los niños nacen perfectos. Eso nos permite afirmar que la naturaleza odia la tolerancia a fallas. Pero no vemos ningún mecanismo explícito para lograrlo. En cambio, la tolerancia cero a fallas surge de mecanismos fundamentales que deben ser muy complicados y extremadamente frágiles. En cualquier caso, la variación registrada a nivel molecular es una fracción ultramicroscópica de todas las variaciones posibles. Lo sabemos porque la mayoría de las mutaciones a nivel molecular son letales. De hecho, la Teoría Neutral de la Evolución Molecular dice que las mutaciones son en su inmensa mayoría letales, pero una fracción menor es neutral, es decir los mutantes se desempeñan igual de bien o de mal que los originales.

Sumando todo, decidimos nuestra ley:

- MicroLisp 0.3 incluye solo el par de corchetes [] como señales para iniciar y finalizar bucles. Un programa para ser sintácticamente correcto debe estar emparejado: el número de señales de abrir debe ser igual al de las de cierre, y un paréntesis de cierre siempre debe emparejar con otro de apertura a su derecha.
- Administrar el contador de bucles es un deber del desarrollador, en este caso de la PG. Advertencia: JS y otros lenguajes también disfrutan de la libertad de modificar el contador de bucles y es la causa de terribles errores.
- Para evitar bucles infinitos, los programas pueden ejecutarse durante un límite predefinido de rondas, por ejemplo, 100.
- Los programas sintácticos incorrectos pueden participar en la recombinación pero no son elegibles para la clonación para generar la generación siguiente.

Tenga en cuenta que nuestras reglas garantizan la ejecución esperada de programas sintácticamente correctos, pero no prohíben todos los flujos ilegales. Esta política permite encontrar agradables sorpresas como el programa $+[+.]$ que suena como un bucle infinito porque no incluye ningún comando decreciente. Sin embargo, se detiene sobre los dígitos porque el 0, la señal del terminal del bucle, se encuentra de forma recurrente junto con incrementos sucesivos. Además, el programa $-[-.]<$ genera 8,7,6,5,4,3,2,1,0. ¿Qué hace el programa $-[.-]<$ sobre dígitos?

Debido a que estamos creando un lenguaje que se está volviendo complicado y cuyos resultados se utilizarán para respaldar conclusiones importantes, debemos ase-

gurarnos de que todo sea como decimos. Por lo tanto, un depurador de lenguaje para MicroLisp 0.3 es una necesidad urgente. Aquí viene uno.

3.6. Editor avanzado de MicroLisp 0.3

El program `MicroLispComputer03.html` presenta un Entorno Integrado de Desarrollo para MicroLisp 0.3 que incluye

- Un editor avanzado que permite la tabulación y los comentarios. Convierte la complejidad en un juego artístico porque una correcta tabulación hace que los programas sean ordenados y legibles por humanos.
- El alfabeto de celda se puede seleccionar para que sea 01 binario, dígitos 0123456789, o dígitos con espacio en blanco 0123456789b, donde la b se puede sustituir por un espacio o por un guión.
- Los símbolos en las celdas de memoria deben interpretarse como caracteres. Sin embargo, en la presente implementación, utilizamos números enteros no negativos como intermediarios.
- El lenguaje de programación es MicroLisp 0.3 que admite bucles.
- Las operaciones de copiar y pegar están habilitadas.
- Los programas se pueden ejecutar. Copie y pegue los programas de esta sección y la siguiente directamente en el panel del editor para probar fácilmente lo que se afirma.
- Para ayudar a depurar, cada paso del Intérprete se publica en la Pila de Ejecución (Execution stack) que debe leerse de abajo hacia arriba.

Este IDE fue desarrollado por el Autor a partir de un programa de Michael Duerinckx l.c., ([9] 2014). Veamos cómo se ven los programas en este editor. Todos los programas de esta subsección se ejecutaron con 16 celdas de memoria y Dígitos como alfabeto de celda.

Ejemplo. El siguiente programa coloca 3 en la celda c0, 4 en la celda c1 y suma los dos números en la celda c2. Los datos se imprimen a continuación: $3+4=7$. Los números se imprimen sin espacios: 347 es la salida.

```
+++ >> +++            Suma 3 a la celda c0 y 3 a la celda c2
<                     Mueve el puntero a la celda c1
++++>++++             Agrega 4 a la celda c1 y agrega 4 a la celda c2
<<                    Mueve el puntero a la celda c0
.>.>.                 Todos los valores se imprimen sin espacios
```

El editor ignorará los comentarios, pero si ha instalado un corrector ortográfico en su sistema, también funcionará aquí (probado en Ubuntu 20.4). Los comentarios no pueden tener ningún símbolo del idioma MicroLisp 0.3, de lo contrario serán interpretados como parte de la programación. Entonces, en sus comentarios, los puntos están

prohibidos. Use en cambio letras mayúsculas. Puede usar aquí la coma pero, tenga cuidado, en otros idiomas similares, la coma es parte del idioma, así que asegúrese de conocer bien los comandos de su idioma.

11 ***Ejercicio.*** *Desarrolle y ejecute un programa en MicroLisp para restar 2 de 7. Los números y la respuesta final deben imprimirse (725).* Answer

12 ***Ejercicio.*** *Componga un programa para Dígitos que calcule* 2×3*, que imprima los números y el resultado. Salida deseada 236.* Respuesta

Desafíos. Primero: Componga un programa para una computadora con 2 celdas de memoria para sumar 3 a 5. Los números y la respuesta final deben imprimirse. Salida esperada = 358. Segundo: Haga lo mismo, pero inserte separadores. Resultado esperado 3 5 8. De acuerdo con su experiencia personal, ¿el tamaño de la memoria es una necesidad o es solo una característica amigable para los humanos?

Advertencia sobre bucles: La repetición de una tarea agrega modificaciones a las modificaciones. Por lo tanto, no se espera generar el mismo resultado para cada ronda. Un bucle o repetidor está delimitado por el par de corchetes []. El contador de bucle es el número que está contenido en la celda que está activa en el momento en que se lee el símbolo [. La modificación del contador de bucles no se realiza automáticamente. En cambio, eso debe hacerlo el propio código que debe escribir el desarrollador.

Calculamos 2×3 como resultado de sumar dos veces 3. ¿En dónde hay un ejemplo de multiplicación genuina?

Ejemplo. La siguiente macro, o programa, usa un ciclo para encontrar 2×3. Recuerde que no se permiten puntos en los comentarios. Solo se imprime el resultado.

```
++              La celda c0 inicia el contador de bucle = 2
[               Comienzo del ciclo
> +++           Moverse  a la celda c1 y agregar 3
< -             Volver a la celda c0 y decrementar el contador
]               Fin del ciclo
>.              Imprimir resultado
```

13 ***Ejercicio.*** *Añada al programa anterior la posibilidad de imprimir los números entrantes y el resultado de su multiplicación.* Respuesta

14 ***Ejercicio.*** *Diseñe un programa para calcular* 2×4*. Los números entrantes y el resultado de su multiplicación deben imprimirse sin espacios (248). Añada comentarios claros. No deben contener ningún símbolo reservado del idioma. Preste especial atención al punto.* Respuesta

Ejemplo. Compongamos un programa que ilustre las posibilidades de MicroLisp 0.3 de ser amigable con los humanos. La multiplicación 2×3 se divide en tres tareas: imprimir el entrante 2, imprimir el entrante 3, multiplicar 2×3 e imprimir el resultado.

Esperamos que el problema sea trivial si reutilizamos una macro anterior. Reutilizar material existente es la *conditio sine qua non* tanto del desarrollo de Software como de la Evolución. La siguiente es nuestra propuesta en la que se asignan tareas a diferentes secciones de memoria:

```
 ++.         Agregue 2 a la celda c0 e imprima el resultado
> +++.       Agregue 3 a la celda c1 e imprima el resultado
>            Moverse a la celda c2

             Se inserta macro para encontrar 2 X 3
             La enumeración de celdas se desplaza a 0

++           La celda c0 inicia el contador = 2
[            Inicio del repetidor
    > +++    Añadir 3 a la celda c1
    < -      Vuelva a la celda c0 y decremente el contador
]            Fin del repetidor
>.           Imprimir resultado
```

Advertencia. Uno podría pensar en desarrollar un lenguaje y editor MicroLisp que permita números enteros como alfabeto celular. Este sueño no es realizable: todos los lenguajes truncan a los números enteros en algún momento, ya sea directamente o por la incapacidad del sistema de manejar números y cadenas extremadamente grandes.

15 ***Ejercicio.*** *Ejecute el programa* $+++[>.++[>.+<-]+.<-]$ *para probar la capacidad de MicroLisp 0.3 de manejar bucles anidados. Reescriba este programa en una forma legible para humanos. Añada comentarios. El programa se detiene y, con dígitos y 8 celdas de memoria, genera 00111234115671. La detención del programa es posible porque en nuestros alfabetos de celdas circulares, el incremento del contador regresa al final a 0.* Respuesta

16 ***Ejercicio.*** *Haga su mejor esfuerzo para desarrollar un programa para la multiplicación* 4×3. *Los valores de entrada y salida deben imprimirse sin espacios. Salida esperada 4312.* Respuesta

Intriga. ¿Es realmente imposible desarrollar un programa MicroLisp 0.3 para calcular sin trucos 4×3 con la respuesta 12?

17 ***Ejercicio.*** *Use bucles para escribir un programa para MicroLisp 0.3, que genere 0-1-2-3-4-5-6-7-8-9. Compare su longitud con la solución sin bucles presentada por el demo correspondiente en el Editor.* Respuesta

3.7. PG con MicroLisp 0.3

Nuestra tarea es cazar una lista de cadenas binarias aleatorias con programas MicroLisp 0.3. Se resuelve con el programa `MicroLispGP03.html`. Nuestra conclusión

es que la mayoría de los programas no producen resultados y eso de debe en general a problemas sintácticos, por lo que el costo de la PG 0.3 es muy alto, como se muestra en el siguiente informe:

```
Costo exponencial de cazar una cadena binaria de longitud x
con MicroLisp 0.3

x          Evolución                  Azar
1          NaNeˆ(NaNx)                NaNeˆ(NaNx)
2          7.14eˆ(2.64x)            4.41eˆ(1.29x)
3          55.7eˆ(1.1x)             0.59eˆ(2.79x)
4        152.75eˆ(0.49x)            1.84eˆ(2.12x)
5         52.33eˆ(1.03x)            1.63eˆ(2.18x)
6         35.13eˆ(1.2x)             2.05eˆ(2.08x)
7         63.54eˆ(0.98x)            3.04eˆ(1.93x)
8         98.23eˆ(0.83x)            6.28eˆ(1.69x)
9         99.38eˆ(0.83x)            9.14eˆ(1.57x)
10        178.4eˆ(0.67x)           11.17eˆ(1.52x)
11       328.98eˆ(0.52x)           10.07eˆ(1.55x)
12       253.09eˆ(0.58x)           13.19eˆ(1.48x)
13       209.02eˆ(0.62x)           15.72eˆ(1.45x)
14       208.73eˆ(0.62x)           16.16eˆ(1.44x)
15       164.18eˆ(0.66x)           17.98eˆ(1.42x)

Repetición:
15       245.16eˆ(0.59x)           6.12eˆ(1.51x)
```

Con respecto a la primera corrida, una computadora de escritorio necesitó media hora para llegar a 12, pero 7 horas adicionales para hacer 13 y 14, y 15 horas más para calcular 15.

Evaluación. La regulación lo es todo en la Biología. Por lo tanto, si la regulación se introduce en un reino vivo por cualquier causa, debe permanecer allí para siempre. Tomando prestada la jerga de la Teoría Evolutiva de Juegos (Cowden [7] 2012), podríamos decir: la regulación es una E3 o Estrategia Evolutiva Estable en el juego de la Evolución vs complejidad. Por eso hemos ampliado nuestro MicroLisp para dejar espacio a la regulación, en nuestro caso a los bucles. El problema es que MicroLisp 0.3 nos muestra que la regulación es demasiado costosa para la Evolución.

18 ***Ejercicio.*** *Represente gráficamente una caminata aleatoria en el espacio de los programas MicroLisp 0.3. Analice los resultados.* *Respuesta*

3.8. El mar del sinsentido

Para reconstruir la geografía del espacio de programas según los resultados del último ejercicio, hagamos la siguiente convención. Los programas sin salida se corresponden con el agua, y los programas con output se corresponden con la tierra. El paisaje emergente es entonces el de un gran mar que es salpicado por unas pocas islas diminutas.

19 ***Ejercicio.*** *Ejecute el programa* `NonsenseSeaEnglish.html` *para verificar que en cualquier lenguaje humano, las oraciones con sentido se conforman como pequeñas islas que están rodeadas por un inmenso mar de sinsentido.*

Dado que los programas sin salida se corresponden con cadenas sin sentido, resulta un cuadro simple: en el espacio de los programas MicroLisp 0.3, el mundo se compone de pequeñas islas de cadenas con sentido que están rodeadas por un inmenso mar de sin sentido.

20 ***Ejercicio.*** *Represente el tapiz de la perfección para la PG sobre MicroLisp 0.3. ¿Considera que la imagen emergente es compatible con la idea del mar del sin sentido?* *Respuesta*

4. EL COSTO DE LAS APROXIMACIONES

Sabemos que el costo en la PG 0.3 de la coincidencia exacta es exponencial. ¿Qué pasa con el costo de la coincidencia aproximada?

Desafío. Estudie el costo de las aproximaciones en PG con MicroLisp 0.3 y discuta las consecuencias. Sugerencia: Siga el procedimiento correspondiente que aparece en GP1.

5. CONCLUSIÓN

Cuando el trabajo es inteligente, las conclusiones son claras y contundentes.

21 ***Ejercicio.*** *Proponga una conclusión con respecto a la pregunta que enfrentamos: ¿Cuál es el costo de la PG con regulación?* *Respuesta*

22 ***Ejercicio.*** *¿Cuál es en su opinión la conclusión más importante del presente trabajo?* *Respuesta*

Respuestas

Ejercicio 3, page 8. Flipp consiste en una computadora con una celda de memoria y un lenguaje para programarlo. Las celdas de memoria guardan un bit, un cero o un uno. El lenguaje tiene dos comandos. El primero es el comando Punto . que imprime el contenido de la celda de memoria. El segundo comando es el signo de exclamación ! que cambia el valor de la memoria e imprime el resultado: 0 se cambia en 1 y viceversa. MicroLisp 0.1 amplía Flipp porque el primero se puede restringir a solo una celda de memoria y porque uno puede expresar los comandos Flipp en MicroLisp 0.1. De hecho, los operadores de impresión son los mismos sobre la primera celda de memoria, y el comando de signo de exclamación sobre bits de Flipp se puede simular en MicroLisp 0.1. como $+$. o como $-$. porque las operaciones aritméticas son cíclicas en MicroLisp 0.1. Así, definimos **MicroLisp 0.0** como el lenguaje que opera sobre una Computadora Tipo Cuaderno y cuyo conjunto de comandos se compone de los dos comandos de MicroLisp 0.1 $+$. que respectivamente agrega uno e imprime.

Ejercicio 4, page 11. Si dividimos los dos costos, obtenemos:
$r = \frac{e^{0,31x}}{e^{0,3x}} = e^{0,01x}$.
For $x = 1400$, $r = e^{14} = 1000000$. Y la desproporción crece exponencialmente con x.

Ejercicio 5, page 11. Estudiamos la tendencia del ajuste exponencial en rangos crecientes de la longitud de entrada. Vemos que después de largas fluctuaciones al principio, el exponente tiende a una cuasi-estabilización. Esto es imposible para los polinomios que necesitan recurrir a grados cada vez mayores del término dominante. La extrapolación a rangos largos de un ajuste exponencial se realiza a través del exponente c. Está en las inmediaciones de 1. Por lo tanto, consideramos que tenemos derecho a afirmar que la PG en todas las versiones de MicroLisp se ejecuta en el régimen exponencial debido a la tendencia universal observada a la cuasi-estabilidad de los exponentes.

Ejercicio 6, page 11. El programa `GPwithMicroLisp01.html` produce sobre el intervalo [1,18) en dos corridas los siguientes resultados:

```
Costo Exponencial de cazar una cadena binaria de largo  x
con MicroLisp 0.1.

x    Evolución          Randomness          Worst case searching
1    NaNe^(NaNx)        NaNe^(NaNx)         NaNe^(NaNx)
2      35e^(0.69x)    10.3e^(0.15x)       0.2e^(2.08x)
3    55.6e^(0.35x)     2.2e^(1.31x)       0.6e^(1.39x)
4      99e^(0x)        2.2e^(1.3x)          1e^(1.11x)
5    73.3e^(0.15x)     2.8e^(1.19x)         1e^(1.11x)
6    32.7e^(0.5x)      2.1e^(1.32x)       0.9e^(1.15x)
7    22.8e^(0.63x)       2e^(1.34x)       0.8e^(1.19x)
8    16.6e^(0.74x)     2.1e^(1.31x)       1.1e^(1.11x)
9    18.3e^(0.71x)     2.7e^(1.24x)       2.1e^(0.9x)
10   16.2e^(0.74x)     1.9e^(1.33x)       1.6e^(0.97x)
11   15.3e^(0.76x)       4e^(1.15x)       1.3e^(1.03x)
12   22.4e^(0.67x)     2.9e^(1.22x)       1.3e^(1.03x)
14   15.2e^(0.75x)     3.2e^(1.2x)          1e^(1.09x)
15   10.4e^(0.82x)     1.9e^(1.3x)        0.8e^(1.13x)
16   13.7e^(0.77x)       2e^(1.29x)       0.9e^(1.1x)


Otra corrida:


1    NaNe^(NaNx)        NaNe^(NaNx)         NaNe^(NaNx)
2    23.3e^(1.1x)       5.5e^(1.47x)          1e^(1.39x)
3    48.5e^(0.55x)     26.9e^(0.28x)       1.6e^(1.04x)
4    40.4e^(0.66x)     10.6e^(0.84x)       2.8e^(0.69x)
5    35.2e^(0.73x)     10.9e^(0.82x)       2.5e^(0.76x)
6    66.1e^(0.46x)     12.3e^(0.77x)         3e^(0.67x)
7    86.8e^(0.36x)     17.7e^(0.64x)       2.7e^(0.72x)
```

8	61.3e^(0.47x)	7.1e^(0.94x)	2.2e^(0.79x)
9	68.2e^(0.44x)	6.7e^(0.96x)	2.3e^(0.77x)
10	92.8e^(0.36x)	8.8e^(0.88x)	1.4e^(0.91x)
11	80.8e^(0.39x)	5.7e^(0.99x)	1.4e^(0.91x)
12	44.7e^(0.53x)	3.9e^(1.08x)	1e^(0.99x)
13	82.8e^(0.4x)	4.1e^(1.07x)	1.2e^(0.95x)
14	44.1e^(0.52x)	2.2e^(1.2x)	0.7e^(1.04x)
15	29.6e^(0.6x)	2.1e^(1.21x)	0.6e^(1.07x)
16	19.2e^(0.67x)	2.5e^(1.18x)	0.6e^(1.09x)
17	21.1e^(0.66x)	2e^(1.21x)	0.7e^(1.06x)

El costo de la búsqueda del peor de los casos es menor que el de la aleatoriedad porque la composición automática de programas utiliza solo dos símbolos, los de MicroLisp 0.0. Concluimos que la PG en MicroLisp 0.1 es cuadráticamente más cara que en Flipp: es para Flipp proporcional a $e^{0,3x}$ y para MicroLisp 0.1 es aproximadamente proporcional a $e^{0,6x}$. Vemos que todo sucede en el régimen exponencial.

En relación a la primera corrida, la forma en la que percibimos el efecto de un costo exponencial fue la siguiente: comenzando con contraseñas de longitud 1, uno llegaba a la longitud 10 casi instantáneamente, luego venían 11, 12 y 13 en 29 segundos. Pero llegar al 14 necesitó 44 minutos, mientras que el 15 y el 16 necesitaron 10 minutos cada uno, y resolver el 17 necesitó 4 horas adicionales. Al considerar la escalada de tiempo que aparece una y otra vez, el Autor siente una fuerte tendencia a pensar que la TE es estúpida. Por eso debo estar en guardia. En primer lugar, la Evolución aplicada da frutos todos los días. Pero por otro lado, la TE es una teoría científica que es falsa, una afirmación que significa que el veredicto por defecto para la relación Evolutiva de cualquier par de especies es negativo y que sólo el Método Científico puede argumentar positivamente sobre casos específicos.

Ejercicio 7, page 11. El programa `MLisp01GPDynamics.html` muestra la dinámica del Proceso Evolutivo que permite que los programas MicroLisp sintetizados por Evolución cacen una contraseña que representa un tablero de ajedrez. Problema encontrado: MicroLisp 0.1 no puede cazar cadenas binarias de longitud 256 porque produce salidas demasiado cortas. Cuándo la complejidad es grande, las secuencias binarias de salida son demasiado cortas para especificar la totalidad del tablero. Hay tres soluciones. Primera: en nuestro programa para una cadena binaria de longitud L, permitimos que los programas MicroLisp 0.1 tuvieran una longitud entre L y 2,5L. Podemos ampliar el rango permitido. Segunda: modificar el alfabeto MicroLisp para aumentar las probabilidades de impresión. Así, en lugar del alfabeto $< +- >$. podemos usar $< +- >$ .. o $< +- >$ ... o $< +- >$ ..... Tercera: recompensar los programas más largos. Nuestra elección fue probar el alfabeto modificado $< +- >$ .... sobre el mismo programa. Funcionó bien: la longitud de las salidas ahora es suficiente, aunque no necesariamente desde la primera generación. Los programas muestran que la dinámica se estanca en un valor frustrante. Sin embargo, hay fluctuaciones debidas a la mera aleatoriedad, y gracias a ellas, no se excluye que el objetivo pueda lograrse.

Ejercicio 8, page 12. Nosotros modificamos el alfabeto de MicroLisp en el programa `GPwithMicroLisp01.html`. En lugar de $< +- >$., usamos $+- <>$ ...., con cuatro comandos de impresión. Sobre el intervalo [1,18) el costo de PG fue $19{,}3e^{0,71x}$, el de la aleatoriedad $22,9e^{0,69x}$ y, en el peor de los casos de búsqueda $2,1e^{0,98x}$. Una repetición sobre el mismo intervalo dio el costo de la PG como $7{,}7e^{0,9x}$, por aleatoriedad

$4{,}3e^{0{,}89x}$, y para la búsqueda en el peor de los casos $0{,}9e^{1{,}07x}$. Nuestra conclusión es que el comportamiento del nuevo alfabeto respecto a la PG no es mejor que el inicial. Pero estamos muy intrigados porque la aleatoriedad parece ser mejor que Evolución. Esto podría ser fruto de las fluctuaciones que siempre están presentes.

Ejercicio 10, page 15. Usamos MicroLisp 0.2 con el subconjunto de comandos $+$. que funciona sobre cadenas binarias. Nuestro programa `MicroLispGPBlank02.html` compara el costo de adivinar una cadena binaria frente a adivinar la misma cadena a la que se ha adjuntado b al final. Para la primera tarea, el alfabeto de células fue 01, mientras que para el segundo fue $01b$. Usamos una matriz de memoria de 200 celdas que probablemente evitaría la interferencia de la circularidad. Como era lo esperado, incluir espacios en blanco es más caro. Verificado para el intervalo [1, 14): el costo de la PG sin regulación resultó ser $21{,}8e^{0{,}66x}$ y con regulación $12{,}8e^{1{,}19x}$. Se observó una diferencia del exponente por encima de $0{,}3$ en todos los subintervalos. Una repetición hecha para el intervalo [1, 18) dio : sin regulación $6{,}7e^{0{,}89x}$, y con regulación $8{,}7e^{1{,}31x}$. En resumen: el costo de introducir la regulación es menos que cuadrático.

Ejercicio 11, page 20. El siguiente programa resta 2 de 7:

```
++++++++.          Agregue 7 a la celda c0, imprima el valor
>>                 Mueva el puntero a la celda c2
+++++++            Sume 7 a la celda c2
<                  Mover el puntero a la celda c1
++.                Agregue 2 a la celda c1, imprima el valor
>                  Moverse a la celda c2
--.                Restar 2 de la celda c2, imprimir el valor
```

Ejercicio 12, page 20.
El siguiente programa calcula 2×3:

```
++.             Imprima 2 desde la celda c0
> +++.          Imprima 3 desde la celda c1
> +++ +++ .     Añada 2 veces 3 a la celda c2 e imprima
```

Ejercicio 13, page 20.
El siguiente programa usa un ciclo para encontrar 2×3. Los números entrantes y el resultado se imprimen sin espacios (236):

```
++.          La celda c0 inicializa el contador de bucles = 2
[            Comienzo del ciclo
> +++.       Moverse a la celda c1 y agregar 3 e imprimir contenido
< -          Volver a la celda c0 y decrementar el contador
]            Fin del ciclo
```

Ejercicio 14, page 20.
El siguiente programa calcula 2×4:

```
++.           La celda c0 inicializa el contador de bucles = 2
[             Comienzo del ciclo
  > ++++.     Moverse a la celda c1 y agregar 4 e imprimir contenido
  < -         Volver a la celda c0 y decrementar el contador
]             Fin del ciclo
```

La salida es 248.

Ejercicio 15, page 21. En una forma amigable y con comentarios, el programa $+++[>.++[>.+<-]+.<-]$ se reescribe así:

```
+++               Bucle externo con contador = 3
[                 Marca de inicio del bucle externo
    >.            Moverse a c1 e imprimir su contenido
    ++            Bucle interno con contador =  2
        [         Marca de inicio del bucle interno
            >.    Moverse a la celda c2 e imprimir contenido
            +     Añadir uno a la celda c2
            <-    volver a c1 y decrementar el contador
        ]         Marca final del bucle interno
    +.            Agregar uno al contador en C1 e imprimir
    <-            Volver a C0 y decrementar el contador
]
```

Output 00111234115671 cuando el alfabeto de celdas es el conjunto de los dígitos y hay 16 celdas en la matriz de memoria.

Ejercicio 16, page 21. Nuestra elección es reutilizar un programa anterior amigable con humanos que muestra que esta característica también es la mejor para potenciar la Evolución:

```
++++.       Agregue 4 a la celda c0 e imprima el resultado
> +++.      Agregue 3 a la celda c1 e imprima el resultado
>           Moverse a la celda c2

            Se inserta macro para encontrar 4 X 3
            La enumeración de celdas se desplaza a 0

++++        La celda c0 initializa el contador = 4
[           Comienzo del repetidor para encontrar 4 x 3
    > +++   Añadir  3 a la celda c1
    < -     Volver a la celda c0 y decrementa el contador
]           Fin del repetidor
>.          Imprimir resultado
```

El resultado es 342. El valor esperado era 3412. La falla se explica por el alfabeto celular que cuenta circularmente de 0 a 9. Observemos que deben estar involucradas varias mutaciones coordinadas para producir un programa nuevo y útil. Así surge la complejidad en la programación: un programa tiene varias funciones que operan coordinadamente y que por lo tanto deben mutar coordinadamente.

Ejercicio 17, page 21. El programa debe ejecutarse con un alfabeto de celdas igual a dígitos con espacios en blanco (o guión) y con 16 celdas de memoria. En primer lugar, proponemos una idea, es decir, un programa sin bucles que haga el trabajo pero que se preste para bucles:

```
.                   Imprimir 0
-.                  Imprimir guión
++.                 Imprimir 1
--.                 Imprimir guión
+++.                Imprimir 2
---.                Imprimir guión
++++.               Imprimir 3
----.               Imprimir guión
+++++.              Imprimir 4
-----.              Imprimir guión
++++++.             Imprimir 5
------.             Imprimir guión
++++++++.           Imprimir 6
-------.            Imprimir guión
++++++++.           Imprimir 7
--------.           Imprimir guión
++++++++++.         Imprimir 8
---------.          Imprimir guión
++++++++++.         Imprimir 9
```

Esta idea exige un bucle para cada línea y si se intenta utilizar un doble bucle para incluir todas las líneas, se ve que sería necesaria más de una celda de memoria. Pero en ese caso, la siguiente idea es más adecuada para bucles:

```
>-<         Moverse a la celda c1 Poner guión Regresar
.           Imprimir 0 desde C0
>.<         Imprimir guión desde C1
+.          Imprimir 1 desde C0
>.<         Imprimir guión desde C1
+.          Imprimir 2 desde C0
>.<         Imprimir guión desde C1
+.          Imprimir 3 desde C0
>.<         Imprimir guión desde C1
+.          Imprimir 4 desde C0
>.<         Imprimir guión desde C1
+.          Imprimir 5 desde C0
>.<         Imprimir guión desde C1
+.          Imprimir 6 desde C0
>.<         Imprimir guión desde C1
```

```
+.          Imprimir 7 desde C0
>.<         Imprimir guión desde C1
+.          Imprimir 8 desde C0
>.<         Imprimir guión desde C1
+.          Imprimir 9 desde C0
```

Esta idea se implementó a continuación con un bucle. Se utilizaron tres celdas. La primero, c0, para mantener el contador del bucle, luego c1 para imprimir números, y c2 para imprimir guiones:

```
>>-         Moverse a la celda c2 y poner   guión
<<          Volver a c0

+++++ ++++  Poner contador de bucle = 9
[           Inicio del bucle
            Imprimir números 123456789
    >       Moverse a la celda c1
    +.      Agregue uno a la celda c1 e imprima
            Imprimir guión
    >.<     Moverse a la celda C2 Imprimir guión Volver a la celda c1
    <-      Volver a la celda c0 y decrementar el contador
]           Fin del ciclo c0
.           Imprimir 0
```

Hemos ilustrado que los bucles pueden hacer que el código de tareas repetitivas sea sensacionalmente más corto: la primera idea emplea 107 comandos, la segunda 49 y su implementación de bucle 25. Por otro lado, el hecho de que hayamos podido diseñar programas complejos con MicroLisp 0.3 nos autoriza a seguir con el siguiente paso: estudiar las posibilidades de la Evolución para desarrollar programas en este lenguaje.

Ejercicio 18, page 22. El programa `MLisp03RandomWalk.html` muestra una caminata aleatoria en el espacio de programas MicroLisp 0.3. Está dominada por programas sin salida que son raramente intercalados por programas con salida.

Ejercicio 20, page 23. El programa `MLisp03EvoOfPerfection.html` nos presenta el Tapiz de la Perfección, un dibujo que muestra la Evolución de la aptitud en la PG sobre MicroLisp 0.3. Nos muestra que la Evolución puede salir del mar del sinsentido para andar frecuentemente entre islas con sentido. Pero tras pocos pasos, menos de 7, se cae al mar del sinsentido. Y vuelta a empezar, vez tras vez. Todo eso es tan caótico que la solución perfecta raramente viene de un buen exponente. Puede venir incluso de un programa que no produce output, a los cuales se les permite participar en la recombinación.

Ejercicio 21, page 23. Hemos aceptado que el coste de la PG sin regulación es exponencial con un exponente menor que 1. No encontramos razón para pensar que la regulación pueda reducir el costo. La causa es que las restricciones de sintaxis superan el ahorro de ensayos que posiblemente ofrecen los bucles. Además, la fracción de cadenas binarias que se prestan para bucles es bajo porque una cadena aleatoria tiene unos y ceros en orden y posiciones aleatorios, por lo que no se espera que se repita ningún patrón a excepción de algunos eventos poco frecuentes. Así, la conclusión general de nuestro trabajo es: la PG con o sin regulación funciona en régimen exponencial.

Ejercicio 22, page 23. En nuestra opinión, ningún resultado individual es trascendente. En cambio, la Evolución de la Perfección, representada por el Tapiz de la Perfección, y el costo exponencial juntos permiten formular una predicción obligatoria: si la Evolución me creó, mi cuerpo debe estar lleno de todo tipo de defectos, malformaciones y disfunciones. Falso.

Bibliography

All cited links were active by 28/II/2022

Referencias

[1] ABZHANOV ARHAT 2010
Darwin's Galápagos finches in modern biology Phil. Trans. R. Soc. B3651001–1007
https://royalsocietypublishing.org/doi/10.1098/rstb.2009.0321 14

[2] REGINA BAILEY 2019
The Structure and Function of a Cell Wall
thoughtCo.
https://www.thoughtco.com/cell-wall-373613 14

[3] KORY BECKER AND JUSTIN GOTTSCHLICH 2017
AI Programmer: Autonomously Creating Software Programs Using Genetic Algorithms
https://arxiv.org/pdf/1709.05703.pdf 6

[4] ADAM GORDON BELL 2021
INTERCAL, YAML, And Other Horrible Programming Languages
https://earthly.dev/blog/intercal-yaml-and-other-horrible-programming-lang 6

[5] BITMERGE 2016
How Turing Machines Work
https://www.youtube.com/watch?v=cDc6Gfo3egk 6

[6] DAVID M. BODINE 2014?
Gene Regulation, nhgri.
https://www.genome.gov/genetics-glossary/Gene-Regulation 13

[7] COWDEN C.C. (2012)
Game Theory, Evoluciónary Stable Strategies and the Evolución of Biological Interactions, Nature NEWS 29 April 2021
https://www.nature.com/scitable/knowledge/library/game-theory-evolutionary- 22

[8] MIKE DAVEY 2010
A Turing Machine - Overview

https://www.youtube.com/watch?v=E3keLeMwfHY
http://aturingmachine.com 6

[9] MICHAEL DUERINCKX 2014
JavaScript Brainfuck interpreter
https://github.com/michd/brainfuck-interpreter-js 6, 19

[10] AMBRO VAN HOOF, ROY PARKER 2002
Messenger RNA Degradation: Beginning at the End, *Current Biology, Volume 12, Issue 8, 2002, Pages R285-R287, ISSN 0960-9822,*
https://www.sciencedirect.com/science/article/pii/S0960982202008023
14

[11] JAMES D. MCCAFFREY 2018
The P" Programming Language
https://jamesmccaffrey.wordpress.com/2018/09/05/the-p-programming-language,
6

[12] THE NOBEL PRIZE IN PHYSIOLOGY OR MEDICINE 1965.
NobelPrize.org. Nobel Prize Outreach AB 2021. Tue. 29 Jun 2021.
https://www.nobelprize.org/prizes/medicine/1965/summary/
12

[13] JOSÉ RODRÍGUEZ 2021 *Falsification of the Evolución ary Theory with Flipp*.
Text without programs:
http://www.ejristos.com/GPfalsifiesEvo/FalsificETwithFlipp.pdf
(English)
http://www.ejristos.com/GPfalsifiesEvo/FalsifTEconFlipp.pdf
(Spanish)
Texts and programs:
http://www.ejristos.com/GPfalsifiesEvo/FlippETfalsification.zip
2, 3, 8

[14] JOSÉ RODRÍGUEZ 2021 *The Darwinian Evolución ary Theory at its face value*.
Text without programs:
https://ejristos.com/dl/DarwinAtFaceValue.pdf
https://ejristos.com/dl/DarwinEnSerio.pdf (Spanish)
Text and programs:
https://ejristos.com/dl/DarwinAtFaceValue.zip 11

[15] TWO-BIT HISTORY 2018
How Lisp Became God's Own Programming Language
https://twobithistory.org/2018/10/14/lisp.html 5

[16] VIRTUAL GENETICS EDUCATION CENTRE 2015?
Gene Expression and Regulation https://www2.le.ac.uk/projects/vgec/highereducation/t
13

[17] CATHERINE ZANDONELLA 2015
A gene that shaped the evolution of Darwin's finches Princeton News February 2015
https://www.princeton.edu/news/2015/02/11/gene-shaped-evolution-darwins-fi
14