¿De qué te sirve escribir?

Eduardo Graells-Garrido[1]

Cuando era estudiante a mediados de los dos mil, me gustaba tanto escribir que incluso planteé a mis padres la idea de abandonar ingeniería y estudiar literatura. En retrospectiva, fue una solicitud ingenua debido a las escasas oportunidades laborales y a que estábamos endeudados con un crédito CORFO. Mi madre todavía recuerda la estrepitosa respuesta de mi padre. Yo no la recuerdo, probablemente reprimí esa noche. Sin embargo, dejé un testimonio, que encontré mientras preparaba este texto. Era un cuento corto que escribí al respecto. Allí describí lo que él me preguntó esa noche: *¿De qué te sirve escribir?*

No supe qué decir. Hoy sabría qué responder, porque como ingeniero y científico me he dedicado a la escritura. Los artículos científicos y las propuestas de Fondecyt que he escrito son las que me permitieron trabajar en la Universidad de Chile, recorrer el mundo presentando mi trabajo y, lo que más valoro, enseñar. Quizás no recuerdo lo que pasó esa noche, pero sí recuerdo que aprendí a *escribir* mi camino en la ingeniería y en la academia.

En ingeniería escribimos muchas cosas. Se nos enseña a escribir informes y demostraciones. Se escriben correos electrónicos, men-

[1] Doctor en Tecnologías de la información, las comunicaciones y los medios audiovisuales por la Universitat Pompeu Fabra. Actualmente es profesor asistente del Departamento de Ciencias de la Computación de la Facultad de Ciencias Físicas y Matemáticas de la Universidad de Chile. egraells@dcc.uchile.cl

sajes en U-cursos (a veces chistosos), confesiones en redes sociales (sí, los profes las leemos…). Incluso hay gente que escribe historias, como las del concurso *Beauchef en 100 palabras*. Y eso solo en lo visible.

De hecho, en el Plan Común de FCFM se enseña a escribir código. Quizás no se le llama explícitamente así, sino programar o desarrollar, pero en la práctica el código también es escritura: necesitamos comprender un proceso (matemático, ingenieril, etc.) y especificarlo a la máquina con un vocabulario y gramática específicos para que esta pueda llevarlo a cabo de manera determinística. Eso me parecía fascinante. Si yo tenía claro cómo se desarrollaba una tarea, una máquina podía ejecutarla y yo podría ver el resultado en la pantalla. Mi motivación para estudiar ingeniería en computación era la programación de videojuegos y, por lo mismo, el poder *ver en pantalla* era importante. *Hello, World*, el primer programa que se suele aprender en cualquier lenguaje consiste en *mostrar en pantalla* un mensaje: «Hola, mundo». Era la demostración de que había elegido bien.

No tengo registros de mi primer «Hola, mundo», pero sí recuerdo que me lo enseñó Jo Piquer en una de las salas B del entonces «Edificio de Computación», hoy Beauchef 851 norte. Era así:

```
class HelloWorld {

    public static void main(String[] args) { System.out.println("Hello, World!");
    }

}
```

Puedo decir «era así» porque ese es el código estándar para un programa en Java, el lenguaje que se enseñaba en primer año en 2001. Java es un lenguaje que prometía «ser escrito una vez y ejecutarse en cualquier lugar múltiples veces», en relación con su capacidad de ejecutar exactamente el mismo código en máquinas diferentes, incluso, si tenían arquitecturas de componentes distintas. «Write once» enfatizaba que se debía escribir, pero, al mismo tiempo, le restaba importancia: solo debes hacerlo una vez. Esa filosofía de Java hacía que su código fuese poco atractivo, casi mecánico. Completamente repetitivo y estructurado.

Quizás por eso decayó rápidamente mi entusiasmo por la carrera. La herramienta que nos enseñaban coartaba la creatividad que quería desarrollar. Sin embargo, después de esa noche, que cambió mi perspectiva al forzarme a continuar por mi camino original, tuve una experiencia que me devolvió la esperanza. Cuando aprendí cómo funciona *quicksort* (Hoare, 1962) me maravillé tanto como cuando descubrí la obra de Borges o de Le Guin, porque *quicksort* no funciona de manera intuitiva. Cuando alguien aprende algoritmos, primero se le enseñan métodos que imitan lo que haría una persona. *quicksort* no lo hace. Es una solución ingeniosa que presenta una visión del mundo diferente a la esperada. Eso me chocó. El código, más allá del resultado de su ejecución, en sí mismo podía ser una fuente de maravilla. Se podía *crear* con él, tal como se puede crear un mundo con una historia. *Programar es escribir*.

Eso me planteó algunas preguntas. Por ejemplo, es común que al programar se trabaje sobre el código de otras personas. A veces solo conocemos la API[2] de ese código: sabemos lo que hace, o lo leemos de su documentación, pero no cómo lo hace: es una caja negra. Es el caso de una «biblioteca» o *software library*. Solo estudiamos el funcionamiento del código de una biblioteca cuando queremos aprender más o cuando queremos hacer algo similar, pero lo suficientemente distinto como para no utilizar el código original. Otras veces no existen bibliotecas completas para lo que necesitamos sino código publicado en la red para su uso con distintos niveles de libertad. En esos casos, solemos hacer *copypaste* y, luego, adaptamos.

Adaptar es una palabra frecuente en programación. Sin embargo, pocas veces un código está tan asociado a quien lo escribió (o a quienes lo escribieron, pues muchas veces su autoría es grupal). Un ejemplo es el siguiente cálculo del inverso de la raíz cuadrada de un número[3], proveniente del código fuente del videojuego *Quake III Arena* (id Software, 1999):

[2] Application Programming Interface: definición de los puntos de entrada en los que podemos llamar a una función de código de un módulo con múltiples funciones.

[3] https://en.wikipedia.org/wiki/Fast_inverse_square_root

```
float Q_rsqrt( float number )
{
    long i; float
    x2, y;
    const float threehalfs = 1.5F;

    x2  = number * 0.5F; number;
    y   = * ( long * ) &y;
    i   =                                   // evil floating point bit   level hacking

    i   = 0x5f3759df - ( i >> 1 );          // what the fuck?

    y   = * ( float * ) &i;

    y   = y * ( threehalfs - ( x2 * y * y ) );   // 1st iteration
//  y   = y * ( threehalfs - ( x2 * y * y ) );   // 2nd iteration, this can   be removed

    return y;
}
```

Entre otras cosas, ese cálculo es necesario para normalizar vectores, una operación común en videojuegos. Es un código reconocible por múltiples aspectos, tanto matemáticos (pues es una iteración de Newton-Raphson) como humanos: ¿la expresión what the fuck?, equivalente a «¿qué chucha?» en Chile, expresa sorpresa debido a lo que está en el texto. ¿Por qué funciona la operación de desplazamiento de bits (i >> 1) y la resta a una «constante mágica» (0x5f3759df)? Se puede revelar el truco leyendo el código y descifrando la precisión de la representación numérica, tal como se puede apreciar una buena historia conectando hilos y descubriendo lo que hay detrás de un giro argumental[4]. Esta función particular ha despertado tanta fascinación que muchos artículos se han dedicado a dilucidar su historia, como Lomont (2003).

Se suele asignar la autoría a John Carmack, pero, incluso si él estuvo involucrado, las ideas tras el código tienen más historia y más nombres detrás. Independientemente de la autoría, este código

[4] Hoy esta operación está implementada por hardware y no es necesario recurrir a esta magia creativa.

o, al menos, sus ideas subyacentes, fue usado explícitamente por cientos de proyectos a los pocos años de haberse publicado y por miles de otros que, incluso sin saberlo, utilizaron la función en sus rutinas.

La literatura no funciona así. La autoría suele ser individual y, a lo más, se habla de influencia, o quizás citas, pero no de reutilización[5]. En la literatura hay *voces*. En el código no. Tal como el nombre de un asteroide puede honrar a quien lo descubrió, un algoritmo puede llamarse en función de quién lo diseñó primero, como el Algoritmo de Dijkstra para encontrar caminos en una red o el Algoritmo de Carmack para graficar sombras en una escena 3D. Sin embargo, nadie ve un algoritmo anónimo y dice «ese estilo de resolución es dijkstriano».

A medida que publiqué mis primeras obras literarias comencé a desarrollar mi voz, que combina computación y narrativa. En mi antología *GAME OVER*, el cuento titulado *Los errores de Sofía* se refiere explícitamente a errores de programación: Sofía es un avatar en un mundo virtual, pero no lo sabe, y a medida que toma decisiones erróneas en su vida aparecen diferentes Excepciones con sus correspondientes mensajes de error, y el *loop* de su vida comienza de nuevo. En mi novela *Lloré sin consuelo sobre tu cuerpo eléctrico*, en el arco final de la obra, aparecen las voces de tres robots/programas (que, en jerga computacional, llamo *dæmonio*). Cada uno utiliza el lenguaje humano, pero no lo entiende realmente ni sabe qué es lo que significa sentir, a pesar de que sí sienten duelo, terror y curiosidad. Aunque esto último parece una paradoja, a veces ni siquiera las personas sabemos lo que significa haber perdido a alguien.

Luego, como editor de textos ajenos en una editorial, comprendí el valor de las voces de quienes he publicado y la responsabilidad de amplificarlas. La autoría tiene un peso que cada

[5] Proyectos como *El Aleph engordado* que, justamente, toma *El Aleph*, de Borges, y lo extiende para convertirlo en una obra diferente, parecen ser más una *performance* sobre la influencia y la autoría que un ejercicio de escritura.

persona carga sobre sus hombros y un buen editor debe cargar a la persona y a su voz e impulsarlas.

Trabajar con textos ajenos me permitió definir mi voz por completo. Descubrí cuál es mi estilo y qué es lo que quiero contar. Siempre lo supe, por supuesto, pero ahora podía delinear hasta dónde llegaba mi voz y dónde comenzaba el canon. Por eso ahora sé que escribir es más que expresarse. Más que contar lo que me parecía importante, también era una manera de entenderme, de desglosar lo que sentía, lo que percibía, lo que anhelaba, lo que me dolía, y llevarlo fuera de mí para poder comprender mis procesos y reapropiarme de ellos, resignificarlos. Entender esa estructura de introspección me dio la capacidad de darle voz a los *dæmonios* de la novela.

Al mismo tiempo, en mi ejercicio profesional, empecé a utilizar Python de modos que ya no producen la maravilla que sí encontré en los viejos códigos de motores de videojuegos, ni en mi propio código ni en el de otras personas. Todavía leo mucho código elegante y legible, pero ya no me inspira. Son palabras especializadas que, simplemente, cumplen tareas. Es código plano. Como dice el *Zen de Python*: *Flat is better than nested* (Peters, 2024) (Plano es mejor que jerárquico).

Estaba convencido de que programar es escribir, pero, ante ese panorama, comencé a cuestionármelo. En el presente, una línea que provoque la expresión «¿qué chucha?», ya no es buen código, porque «no sigue buenas prácticas». El *WTF* es cosa del pasado (el de asombro, al menos; todavía están los causados por código mal escrito), no por corrección lingüística o sintáctica, sino porque el código moderno permite expresar de manera directa muchas operaciones e ideas, sin que para ello sea importante la creatividad, a diferencia de lo que ocurría anteriormente, cuando dicha creatividad era, incluso, más relevante que programar: era necesario *hackear*. El *hardware* disponible es tan potente que quienes programan dejaron de darle importancia al manejo de recursos. La memoria RAM sobra, los ciclos CPU también. Hoy se hacen aplicaciones en Javascript mediante el *framework* Electron, que ha democratizado la creación de utilidades: muchas personas aprendieron Javascript y pueden

ver directamente en la pantalla el resultado de lo que hacen, sin tener que preocuparse de utilizar bien el *hardware* o configurar el sistema gráfico. Son aplicaciones preciosas en su envoltorio, pero aburridísimas por dentro, en lo que a código se refiere.

Llegué al punto de mi vida en que el placer de programar disminuía mientras el de narrar crecía. A pesar de que ambas actividades recurren a los mismos conceptos (lenguaje, vocabulario, gramática, sintaxis, audiencia, organización) y a las mismas operaciones mecánicas (tipear en una pantalla y almacenar en archivos de texto, leer el texto de otras personas, reflexionar), las áreas del cerebro que se activan al programar no son las mismas que al escribir texto de lenguaje natural (Krueger et al., 2020). Mi gran motivación había partido con un supuesto erróneo. Sin embargo, ese error fue determinante en el camino que terminé recorriendo.

Pero en el presente, la irrupción de los grandes modelos de lenguaje (LLM en inglés, *Large Language Models*), también llamados «inteligencia artificial», ha demostrado que pueden programar a partir de lenguaje natural. A pesar de que un LLM puede generar un *software* desde cero a partir de un conjunto de instrucciones escritas, esta vez no se programa para que la máquina actúe. Hay varias diferencias. Lo que entrega el LLM no es determinístico: no solo depende del LLM, sino también de lo que diga el azar. No se obtiene la misma respuesta dos veces (¡qué borgiano es eso, un libro que siempre es diferente, como los recuerdos!). Entonces, el actuar de la máquina no es el de antes. Es indirecto: especificamos una instrucción o *prompt*, el LLM genera código que podría satisfacer la instrucción y ejecutarse a la vieja usanza. Ese código generado tiene «un buen lejos» y un «cerca» decente; es difícil encontrarle las pifias y hacerlo requiere tiempo y *expertise*. Ahora cualquiera puede generar código (siempre y cuando tenga acceso a un LLM). No es necesario aprender la sintaxis del lenguaje o el uso de bibliotecas específicas, por muy oscuras que fueran. Los LLM funcionan muy bien con los lenguajes populares como Javascript o Python porque hay mucho código disponible en línea, y lo han utilizado para ajustar el modelo (o «aprender»). De hecho, he explorado sus posibilidades y he implementado sistemas completos. Un LLM implementó mi tesis de magíster, que me tomó dos años, en solo seis minutos.

Pero ¿puede escribir un LLM? Sin duda es capaz de generar un texto narrativo competente. La web está llena de esos textos, los vemos día a día. Además, los LLM se han alimentado de textos de alta calidad (artículos científicos, textos narrativos editados, etc.), a veces sin permiso. Lo que es natural en el código está prohibido en el texto humano. Ahora bien, la máquina nunca sentirá lo que escribe; eso solo puede suceder en la ficción. Aunque un LLM conoce las palabras y sus probabilidades mucho más que cualquier ser humano, nunca conocerá el peso de una palabra ni por qué alguien la eligió. Esta limitación insalvable no es tan relevante si no se necesita emotividad o maravilla. Cuando necesitas escribir un algoritmo o solucionar un problema fundamental de las matemáticas, un LLM podrá apoyarte. Esto ha sido confirmado por personas como Donald Knuth (Knuth, 2026) y Terence Tao (Wong, 2026), dos figuras brillantes de la computación y las matemáticas. También han descubierto que un modelo no puede lograrlo por su cuenta. Necesita ser guiado. Y no es cualquier guía, sino la de alguien con la experiencia de haber resuelto problemas y, sobre todo, de haber comunicado soluciones y maneras de llegar a ellas. Es decir, alguien capaz de leer y escribir lo que se hace en ingeniería.

Eso no es fácil. Conté que Claude Opus implementó mi tesis en seis minutos, pero ¿cuáles fueron los *prompts* que le permitieron lograrlo? ¿Cuál fue el contexto sobre el cual trabajó el LLM? No lo hizo desde cero, sino que existía un repositorio de código previo y de apuntes metodológicos. Entonces, cuando le pedí que implementara una técnica específica de computación gráfica, el LLM siguió convenciones preestablecidas. Me sorprendió que solo con el nombre de la técnica pudiera crear un código funcional en ese contexto que le entregué. Sin embargo, hay un pero: el primer código generó una imagen que se acercaba a lo que yo necesitaba, pero que era notoriamente errónea. Más que algún detalle a corregir, la imagen evidenció un error de comprensión del LLM respecto al significado de la petición. Una metáfora útil para explicar este fenómeno sería que, en vez de armar un rompecabezas con piezas que se conectan entre sí, las piezas no se conectaban de manera exacta, sino que se podían amontonar. Es curioso que la máquina, al entregarme esa solución errónea, me dijera que funcionaba de

manera impecable. Resistí las ganas de decirle que valía callampa la solución (me encantan las situaciones humorísticas que se pueden dar) y, en vez de eso, le indiqué explícitamente en qué se había equivocado y el LLM propuso cómo corregir esos errores. A la cuarta tanda el problema ya estaba resuelto y yo estaba satisfecho. Cuatro tandas en seis minutos. Me impresionó: por primera vez pensé que una máquina podría hacer algo mejor que yo.

Digo «impresionó» en lugar de «maravilló», porque pronto entendí que no estaba comunicándome con un humano y que mi conocimiento es valioso. Es lo que permite no caer en la ilusión de que la máquina sabe lo que hace y también lo que permite guiarla en la dirección de lo que necesitamos al especificar exactamente dónde se equivocó. Todo el sufrimiento propio de desarrollar una tesis de posgrado hecha a la antigua mantenía ese valor, porque sin esos años yo no habría podido escribir los *prompts* de manera efectiva.

Como profesor, estoy en un escenario al que enfrento con incertidumbre. Las personas con perfil *senior* pueden hacer maravillas con estas herramientas y potencian su labor; las *junior* (como quienes se han titulado recientemente), no. Una máquina puede producir mejor y más código, y más rápido. El problema no es que sean *junior*, sino que no tienen la experiencia suficiente para poder leer códigos ajenos o para escribir lo que necesitan desde un punto de vista experto y creativo. Y, dado un LLM, es más difícil que adquieran esa experiencia. Porque el LLM ya la adquirió leyendo textos ajenos y puede producir en pocos minutos lo que a alguien le tomaría incluso dos años. Ya asumí que el código producido por un LLM es mejor que el mío en múltiples aspectos. Uno de ellos es que toma rutas o hace maniobras que yo no habría sido capaz de realizar. ¿Por qué no me sorprende, entonces? Porque es el siguiente paso lógico. Lo que hace la máquina es algo que yo hubiese podido hacer con el tiempo necesario para estudiar, leer, iterar y ejecutar.

¿Sucede lo mismo con textos narrativos? Comencé con la premisa de que programar es escribir, pero la realidad es diferente hoy. Lo que hace un programa puede despertar asombro y maravilla, incluso el proceso para llegar a su código, pero no el código mismo.

No hay subjetividad en la lectura de un código. En cambio, quien lee un texto narrativo no es solamente un espectador, menos aún un operador. Es alguien que pone de su parte en la interpretación del texto para construir el resultado en su imaginación. Esa subjetividad no podrá ser imitada porque un LLM necesita optimizar su resultado y no sabemos siquiera si la subjetividad es una función. No puede tener como *input* la imaginación de una persona. Ya hay experimentos que han llegado a esa conclusión con LLM modernos (Marco et al., 2024), y dudo que cambie en el futuro. Por definición, un LLM genera el promedio de lo que ingirió como texto de entrenamiento (Chiang, 2023). Los promedios nunca destacan.

En mi docencia de computación quiero fomentar la cultura y la imaginación. Quienes cultiven su imaginación mediante la lectura y la escritura ejercerán mejor la ingeniería, porque entenderán el mundo y las subjetividades de quienes lo habitan y podrán comunicarse tanto con esas personas como con los LLM u otros sistemas futuros.

Si tuviera un hijo o hija y me contara que quiere estudiar ingeniería, yo le diría: *¿De qué te sirve hacer lo mismo que hacen las máquinas?*

No sería una pregunta retórica. Yo le ayudaría a elaborar una respuesta que le permita decidir, porque creo que hay muchas oportunidades para la ingeniería. Programar no es escribir, aunque sí tiene la capacidad de maravillar y de cambiar el mundo. Si desarrollas tu comprensión lectora, tu reflexión y tu escritura para humanos, entonces tu programación también mejorará, incluso, si lo haces a través de una máquina que sabe de todo, pero desconoce lo que sabes y sientes tú.

No tendré hijos por decisión propia. Cuando escribo, lo hago desde la posición de alguien cuyo paso por este mundo no dejará otro rastro que su código y sus palabras. Mi padre falleció hace diez años y todavía me duele el no poder compartir con él algo que me maraville. En cambio, si le entrego a un LLM un texto en el que expreso mis sentimientos, en vez de maravillarse me dirá que hay que

mostrar más, contar menos («Show, don't tell»). Buscará acercarme al texto promedio, porque eso es lo que hace. Pero yo no quiero ser promedio ni ser impecable; quiero que el texto se lleve mi dolor al expresar mi imperfección, al contar lo que siento. No es algo que se pueda mostrar. Solo se puede contar y esperar una reacción del otro lado de la pantalla. Hoy, el código en Python o JavaScript es útil y elegante, pero ya no asombra. «Flat is better than nested». «Muestra resultados, no cuentes cómo se hizo (porque ya no importa)». No hay *dæmonios* esperando detrás del código.

Referencias bibliográficas.

Chiang, T. (2023). *ChatGPT Is a Blurry JPEG of the Web — The New Yorker*. https://www.newyorker.co m/tech/annals-of-technology/chatgpt-is-a-blurry-jpeg-of-the-web.

Hoare, C. A. (1962). Quicksort. *The computer journal*, *5*(1), 10–16.

Knuth, D. (2026). *Claude's Cycles*. https://www-cs-faculty.stanford.edu/~knuth/papers/claude-cycles.p df.

Krueger, R., Huang, Y., Liu, X., Santander, T., Weimer, W., & Leach, K. (2020). Neurological divide: An fMRI study of prose and code writing. *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, 678–690.

Lomont, C. (2003). Fast inverse square root. *Technical Report*, *32*, 44–46.

Marco, G., Gonzalo, J., Mateo-Girona, M. T., & Santos, R. D. C. (2024). Pron vs prompt: can large language models already challenge a world-class fiction author at creative text writing? *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, 19654–19670.

Peters, T. (2024). *PEP 20 – The Zen of Python*. https://peps.python.org/pep-0020/.

Wong, M. (2026). *The Edge of Mathematics — theatlantic.com*. https://www.theatlantic.com/technology

/2026/02/ai-math-terrance-tao/686107/.