Semántica computacional*

Camilo Chacón Sartori
camilochs@gmail.com

7 de mayo de 2022

1. Introducción

Para tratar la semántica es, primordial y relevante, primero mencionar la importancia de los lenguajes de programación dentro de la ciencia de la computación, para esto, Turner dice lo siguiente:

> La ciencia de la computación es una disciplina dominada por lenguajes. [...] En particular, sin lenguajes de programación, las máquinas físicas, que algunos defienden como el corazón de la disciplina, serían dispositivos inservibles. La programación le da vida a las máquinas. (Turner, 2018)

Ahora bien, esta afirmación es incompleta. Pues se debería añadir algo más: sin programadores que creen, modifiquen y mantengan a los lenguajes de programación, estos también serían inservibles. Un lenguaje de programación es una herramienta para el uso de los programadores al igual que un instrumento es una herramienta para el uso de los músicos. Los dos pueden crear piezas que son admiradas por otras personas. El primero a través de programas que resuelven una necesidad tecnológica; y el segundo para el disfrute de sus oyentes.

Por otro lado, los lenguajes de programación y, en particular los modernos y populares, no son inmutables, están inmersos en una constante evolución para dar respuestas a nuevas problemáticas tecnologías que, parece ser, no disminuyen con el paso del tiempo. Una característica que, no es extrañar, comparten con casi todo software[1].

Cada lenguaje de programación está compuesto por dos componentes: sintaxis y semántica. La sintaxis es un conjunto de reglas que definen las combinaciones de símbolos, es decir, de cualquier tipo de carácter, por ejemplo, un dígito o letra. La sintaxis se describe con una gramática (libre de contexto) que

*Este documento esta bajo licencia CC-BY-NC 4.0 International license. La primera versión de este texto surgió en mayo del 2021.

[1] Un software es un conjunto de programas computacionales.

se puede analizar usando analizadores sintácticos[2] como: LL(k) y LR(k), para, posteriormente, construir un conjunto de reglas sintácticas según la gramática definida.

Hay dos conceptos claves en la semántica: significado y referencia. La primera sección de este artículo comienzo explicado qué es el significado, luego continuo con la referencia. Finalmente, en la última sección, añado un ejemplo de un problema insoluble de la semántica computacional.

2. Significado

Existen diversos lenguajes de programación los cuales tienen en común una principal diferencia: su sintaxis; otra, es que muchos de ellos son Turing-completos, o sea, pueden simular el mecanismo de una máquina de Turing. Por ello, cada uno sería capaz de resolver un mismo problema, si y solo si este es computable. A su vez, en la literatura se suelen agrupar por «paradigmas» que hace entrever un modelo de representación de la computación; con sintaxis más simple o no; con facilidad de resolver un tipo de problemas sobre otros; y la que nos importa más en este apartado, con diferentes grados de complejidad para conocer su significado.

Supongamos que tenemos la siguiente función[3] f:

```
function f() then
    return a() + b()
end
```

Que dice:

> Cuando invocamos la función f está retorna la suma entre la función $a()$ y $b()$.

Que vendría siendo lo mismo si lo expresamos como objetos matemáticos: $f \rightarrow a + b$. Pero conocer la sintaxis de un lenguaje de programación no es suficiente. ¿Qué valores retornan las funciones $a()$ y $b()$? ¿Son estos números o caracteres para concatenar, u otro tipo de estructura computacional? Claramente con solo conocer su sintaxis y comprobar que es válida, no es suficiente, no nos ayuda para saber que *hace* el código. Para entender su significado necesitamos de la *semántica*.

La semántica es la forma de entender el significado y propósito de un programa computacional. Así, como los filósofos desde un espectro general han buscado una teoría del significado para asignar contenido semántico a las expresiones de un lenguaje natural, en computación, la búsqueda es similar pero dentro de otro contexto, por ello, con otras restricciones.

Por ejemplo, si tenemos dos funciones o procedimientos A y B. La primera se define de la siguiente forma:

[2] Para una introducción, simple y amena, sobre estas cuestiones puede remitirse al libro sobre compiladores escrito por Niklaus Wirth (creador de Pascal, Modula y Oberon): https://people.inf.ethz.ch/wirth/CompilerConstruction/CompilerConstruction1.pdf

[3] En este documento cuando se escriba «función», es lo mismo que decir: método o procedimiento.

```
function A() then
    y = y + 1
    x = 10
    return y + x
 end
```

Donde se aplican dos asignaciones de variables: x e y. Pero asumamos que y es una variable global, es decir, fue definida en otro lugar diferente a la función A, eso significa que, el valor de retorno podría variar en cada ejecución del programa. A esto se le conoce como efecto lateral (*side-effect*). Muchos lenguajes de programación permite esto, y por ello, se vuelve más difícil realizar una validación semántica. ¿Por qué? El valor de y es incierto y depende de la asignación realizada en otro lugar del programa y no, como es de suponer, dentro de A. Por tanto, x tiene asociado el valor 10, pero y es dinámica en tiempo de ejecución, así, $x + y$ no se puede conocer *a priori* sin la semántica.

La segunda, B, en cambio, se define así:

```
function B() then
    y = 10
    x = 10
    return y + x
end
```

Como se puede apreciar esta función evita el *side-effect* ya que, las variables x e y se asignan dentro de la función B y no hay, por tanto, una referencia a un variable que *podría* existir en un contexto global. A este tipo de función se le conoce como pura, dado que siempre que se ejecute retornará el mismo valor, en este caso: 20. Ahora bien, esto tiene algunas implicaciones. Si la función A tuviera definida la variable global y como 9 y no sufre otros cambios en otro lugar excepto dentro la función A, entonces A y B al ejecutarse (independiente de la cantidad de veces) serían *idénticos*. Así, nos damos cuenta de que un programa podría tener diferentes sintaxis, pero sus resultados y comportamiento serían idénticos semánticamente. Es aquí una de las razones del porqué la importancia de la semántica. Si podemos detectar que dos o más programas difieren entres ellos

Un lenguaje de programación está compuesto de diversas estructuras: funciones, variables, bucles, asignaciones, etc., todas cuáles se le puede asignar un valor semántico. Estas puedes llegar a ser más complejas que un lenguaje natural por su naturaleza abstracta y aún más, si en ella existen *side-effect* o sufre del problema de la parada (es decir, indecibilidad)[4]. Algunos ejemplos que dificultan trabajar en la búsqueda de un significado en los programas:

- Los valores de entrada y salida, por ejemplo, si un usuario ingresa su edad en un formulario web, a priori, el valor que estará asignado a la variable que

[4]Una pequeña aclaración: un lenguaje natural es complejo porque es una «máquina» de crear ambigüedades, pero con la dificultad de los lenguajes formales, en concreto, de los lenguajes de programación, me refiero a una complejidad desde el punto de vista humano a código. Entender miles de líneas de código requeriré un enorme esfuerzo sintáctico y semántico, por sobre, a leer, por ejemplo una novela escrita en tu lengua nativa.

contenga su edad y, no es posible conocerlo antes de ejecutar el programa, por ello, solo es posible ser evaluado en tiempo de ejecución.

- Si una expresión lógica que controla a un bucle siempre es evaluada como verdadera, esto asumiría que bucle nunca se detendría y, a su vez, estaríamos frente a un problema indecidible.
- La auto generación de código puede extender la funcionalidad de un programa dinámicamente, es decir, aumentar su capacidad para resolver nuevos problemas según algún evento que ocurra en el tiempo. Esto conlleva una dificultad para encontrar su significado.
- El paralelismo presenta retos interesantes, pues, una misma función puede estar ejecutándose en distintos núcleos disponibles de una CPU, con comportamiento diferentes en el caso de que realicen operaciones de coordinación de datos, esto hace que no sea simple de prever su comportamiento.

La semántica en los lenguajes de programación ya tienen varias décadas de historia, y, existen múltiples enfoques para estudiarla: operacional, formal y axiomática[5]. Asimismo, cabe señalar que, la semántica en los lenguajes ha hecho de mejor aliado a un modelo computacional que proporciona un mejor manejo y control sobre la computación, evitando los *side-effect*, nos referimos a la computación funcional (o para efectos técnicos, conocida como: «programación funcional»)[6].

3. Referencia

El concepto de «referencia» significa muchas cosas en la programación. Por un lado, puede indicar una asociación entre una dirección de memoria con un valor, desde el punto de vista de como se almacenan los datos en un computador; y por otro, sobre una propiedad que posee cualquier programa, y estas, se dividen en dos: referencia «transparente» y «opaca». A esta última, nosotros nos enfocaremos en la primera, transparente, la que nos permite construir *mejores* programas.

Este concepto y otros más, han sido desarrollados desde trabajos que estudian el significado de los lenguajes naturales. Por ejemplo, W. V. Quine, filosofó británico de corriente analítica popularizo el término «transparencia referencial»[7] para indicar lo siguiente: «siempre que una ocurrencia de un término singular, t, es puramente referencial en un término o sentencia $\psi(t)$, es también puramente referencial en el término o sentencia continente $\phi(\psi([t]))$» (Quine,

[5] Para más detalles puede revisar el artículo de David A. Schmidt: http://people.cs.ksu.edu/~schmidt/705a/Lectures/chapter.pdf

[6] Véase mi libro sobre el tema: «Computación y programación funcional», publicado en el 2021, editorial Marcombo.

[7] Quine cita al trabajo de Russell y Whitehead «Principia Mathematica» como los primeros en usar este término.

1960), esto quiere decir que, un término singular o sentencia dentro de una sentencia puede ser sustituido por otro sin perder su significado, Quine entonces, dice que una construcción tiene una referencia transparente si es puramente referencial. El caso contrario, por ende es, una referencia opaca.

Años después, en el artículo de C. Strachey «Fundamental Concepts in Programming Languages»[8] de 1967, presentaría este término en el ámbito de la programación citando a Quine como influencia (Strachey, 2000). A todo ello, esto traería grandes implicaciones en los lenguajes de programación, ya que, esta propiedad demostraría ser de utilidad para desarrollar software y, lo sabrían aprovechar, en particular, los lenguajes de la familia funcional. Ahora explicaremos el porqué.

Primero, su contraste, una referencia opaca es, la propiedad de cualquier función dentro de un programa que no puede ser intercambiada o sustituible por otra función, es decir, en caso de una sustitución su comportamiento no se mantendría intacto. Si tenemos, por ejemplo, un programa P con dos funciones OA y OB:

```
function OA(a: Integer, b: Integer) then
  return a + b
end
```

```
function OB(a: Integer, b: Integer) then
  return a - b
end
```

Estas funciones no son transparente porque si se invoca la función $OA(2,1) \rightarrow 3$ y la intercambio por OB, entonces el resultado sería: $OB(2,1) \rightarrow 1$. Así, ambos resultados son diferentes para los mismos argumentos de la invocación de cada función. Entonces, el programa se comportaría de manera diferente en caso de aplicar una substitución entre ambas funciones.

Segundo, y más importante y nuclear, la referencia transparente (o transparencia referencial) es la propiedad que permite a una función TA ser intercambiada por otra función TB, sin afectar ni alterar, el comportamiento del programa.

```
function TA(a: Integer, b: Integer) then
  return a + b
end
```

```
function TB(a: Integer, b: Integer) then
    c: Integer =  a + b
    return c
end
```

Estas dos funciones son sintácticamente diferentes, además, en su comportamiento dinámico (tiempo de ejecución), la función TB ocupa más espacio extra de memoria por el uso de una nueva variable c, que, no existe en TA. Pero, lo importante aquí es que, si sustituimos TA por TB o TB por TA, el comportamiento del programa será similar y, para ser más específico, en tiempo

[8] Reeditado en el año 2000.

de ejecución. Algo fundamental, a diferencia de la referencia opaca, es que esta propiedad necesita de funciones puras (equivalente a un término singular o sentencia que es puramente referencial). Una función pura es, en términos simples, una función que dado los mismos argumentos debe retornar el mismo resultado.

¿Cuál es la utilidad? Desde una visión *pragmática* es, sin duda, fundamental por las siguientes razones: (1) facilidad para realización, modificación y expansión de pruebas del software; (2) mantenibilidad general, por ello, de todo el software en su conjunto; (3) obliga al sujeto que desarrolla añadir funciones pequeñas y manejable, así, se traduce en una menor complejidad del software.

Una función debe resolver solo una problemática. Un error común es querer resolver múltiples problemas dentro de una misma función, esto es, por así decirlo, uno de los principales errores a la hora de construir software. Todo ello, porque aumenta la posibilidad de *bugs* que, a priori, no son detectados. Incremento de complejidad. Aplicando referencia transparente a nuestras funciones o métodos nos aseguramos de que podemos mantener, escalar y flexibilizar nuestro sistema.

Aunque no crea el lector que esto significa que el programador estará exento de toda dificultad. Evaluar, analizar, contrastar siguen siendo las principales características de una persona bien formada en computación.

4. ¿Lenguaje de programación universal?

En este apartado daré un bosquejo sobre una idea que, ha creado una cierta curiosidad y en algunos casos, una necesidad para simplificar la programación. Esta idea es la de tener un solo lenguaje de programación. Un lenguaje de programación universal que nos permita resolver cualquier problema posible en la actualidad. Para la cual, mostraré algunas objeciones sobre esto.

Para esta empresa, usaré un ejemplo. Supongamos que existe una hábil programadora que tiene la audaz idea de diseñar e implementar un lenguaje de programación universal, el cual reemplazaría a todos los existentes, así, este lenguaje en cierta medida unificaría todos los paradigmas de la programación en uno solo. Prometiendo desaparecer, eliminar y borrar todo rastro de cualquier tipo de disputa, fanatismo y predilección sobre un lenguaje u otro (como dicho sea de paso, ocurre en la actualidad).

Para aquello, ella comienza con dos pasos fundamentales:

- Estudiar los principales paradigma de la programación junto a un lenguaje que lo represente de la manera más exacta.
- Conocer una amplia variedad de problemas que surgen en distintas entidades humanas, ya sea: de diversas industrias privadas y públicas, como también entornos académicos (investigación). Porque ella sabe que, un tipo de problema puede ser resuelto de manera más adecuada con uno o más paradigmas.

Ahora ella se dispone a comenzar a diseñar la gramática de este lenguaje universal. Por ende, se da cuenta de que el primer punto solo es cuestión de

tiempo para comprender una amplia gama de lenguajes asociados a distintos paradigmas, no logrará ni menos es su intención ser experta en ellos, pero si sabe que puede alcanzar un nivel aceptable y decente. En el segundo punto, ella da por hecho que una sintaxis y semántica tiende a ser más precisa para trabajar con un tipo de problema.

Pues bien, mientras trabaja en el segundo punto, comienza a darse cuenta de que cuando anota cada nuevo tipo de problema a una lista, esta comienza a crecer, algunos de ellos tratan de: concurrencia, creación de animaciones visuales, manipular componentes de bajo nivel (registros, espacio de memoria), análisis de datos, manipulación de archivos de audio, extracción de información, etc., sigue escribiendo y no parece que fuera a terminar nunca, ahora bien, ella preveía aquello, por lo mismo, comienza una labor de agregación sobre dicha lista, para entonces, intentar crear generalizaciones de problemas para reducir la lista en un conjunto pequeño. Una meta-lista.

Es aquí donde se percata de que su empresa fracasa pues a medida que va creando agregaciones todavía encuentra nuevos tipos de problemas. ¿Por qué? Ya que los problemas son humanos, son sociales, es decir, varían en el tiempo según las circunstancias y necesidades actuales. Es por eso que, lenguajes naturales, como el español o inglés, no son lo mismo que hace cinco siglos, han cambiado, han evolucionado, algunas palabras quedaron en desuso y nuevas se han añadido. Pues, lo mismo ocurre con los problemas que se puede resolver computacionalmente, son diversos y muchos de ellos no los conocemos todavía por la imposibilidad de conocerlos, ya que, quizás no son un problema en nuestro tiempo.

Un ejemplo de un problema actual es el incremento de datos, antes, dos décadas, por ejemplo, no era posible analizar grandes cantidades de datos, pues los que se generaba eran pocos y tampoco teníamos el hardware preparado para dicha labor. En la actualidad sí lo tenemos, sumado a la aparición de nuevas técnicas que nos entrega el *Big Data* y el *Data Science*.

En suma, la idea del lenguaje universal no es que sea imposible de realizar, es pues, carente de *sentido*. Por las siguientes objeciones:

1. Su gramática tendría que estar constantemente evolucionando lo que, lo haría cada vez más complejo de aprender y usar.

2. Por (1) sería extremadamente difícil crear una compatibilidad retroactiva con programas escritos en versiones anteriores. Sistemas legados.

3. Por (1) y (2) haría imposible que alguien no pretenda crear un lenguaje nuevo, simple y pequeño, para tratar con un nuevo tipo de problema[9].

Existen lenguajes que son multi-paradigmas en la actualidad, algunos cuáles pueden resolver muchos problemas en entornos diferentes y que, exigen, un arduo esfuerzo para dominarlos, por ejemplo, C++, Java y Scala. Todo ellos,

[9] De hecho, hay un incremento en los lenguajes de programación de dominio específicos para tratar con problemas particulares; producto de que el software está cada vez más presente en nuestras vidas.

hoy en día, siguen incorporando nuevas características para dar respuesta a las necesidades actuales que no disminuyen sino que incrementan.

5. Conclusión

La semántica, dicho de manera sucinta, nos amplía la visión sobre las posibilidades de los lenguajes de programación. Nos enseña, en definitiva, lo qué es un programa. ¿Cómo podríamos omitirla?

Referencias

Turner, R. (2018). *Computational Artifacts: Towards a Philosophy of Computer Science*. Springer Berlin Heidelberg.

Quine, W. V. O. (1960). *Word & Object*. MIT Press.

Strachey, C. (2000). Fundamental Concepts in Programming Languages. *Higher Order Symbol. Comput.*, *13*(1–2), 11–49. https://doi.org/10.1023/A:1010000313106