Aprendiendo a desarrollar un intérprete de un lenguaje de programación funcional

Lucas Spigariol[1234], Juan Bono[13], Francisco Sanchez Guijarro[1]
[1]Universidad Tecnológica Nacional
[2]Universidad Nacional de Gral San Martín
[3]Universidad de Buenos Aires
[4]Universidad Nacional de Hurlingham
{lspigariol, juanbono94, franleplant}@gmail.com

Abstract. Se presenta una herramienta de software, cuyo desarrollo está en progreso, que tiene como objetivo facilitar la comprensión del proceso de interpretación de un lenguaje de programación de alto nivel. Su elemento central es un intérprete propiamente dicho y está articulado con una serie de componentes gráficos que permiten visualizar las estructuras intermedias y hacer un seguimiento de los procesos internos que va realizando dicho intérprete, con un sentido pedagógico. Se apunta a un estudiante inicial que quiera aprender sobre el funcionamiento interno de los lenguajes y de la teoría que lo sustenta. Se permite también un uso avanzado, donde ya conociendo el intérprete, el estudiante pueda intervenir modificando o extendiendo la definición del mismo lenguaje.

Keywords: Intérpretes, Lenguajes de programación, Didáctica de la programación, Software educativo.

1 Presentación

El objetivo del software educativo en el que se centra el presente trabajo es facilitar la comprensión del proceso de interpretación de un lenguaje de programación de alto nivel. La principal motivación es la constatación de las dificultades de los estudiantes de comprender y poner en práctica los conceptos involucrados en el funcionamiento interno de los lenguajes de programación y la confianza en que una herramienta de software apropiada puede facilitar el proceso de aprendizaje.

Se trata de un trabajo en progreso, en el marco de un proyecto de investigación, que comienza con un diagnóstico de la situación de enseñanza y aprendizaje en relación a la temática en base al que se plantea el diseño y desarrollo de un software educativo.

Actualmente, se encuentra muy avanzado, al punto de contar con una versión funcionando del software que si bien presenta algunos aspectos pendientes se encuentra disponible para ser utilizada por estudiantes en situaciones concretas.

El diagnóstico realizado permitió identificar los aspectos principales a abordar pedagógicamente y definir el alcance y enfoque de la herramienta de software a desarrollar. Entre ellos, se destacan:

- Dificultades en la comprensión de cada uno de los procesos internos que se realizan sobre el código y del funcionamiento de manera integral.
- Necesidad de un soporte concreto para articular teoría y práctica.
- Importancia de la visualización de los procesos y estructuras de datos.

La etapa de diseño y desarrollo partió de elegir *Lisp* como lenguaje a interpretar, valorando que ya de por sí presenta un modelo relativamente simple y con la flexibilidad suficiente para seleccionar o adaptar diferentes aspectos.

Apuntando a un perfil de estudiante de las materias iniciales de carreras de informática, la herramienta consta como componente central de un intérprete propiamente dicho del lenguaje *Lisp*. La interacción básica se da mediante un editor en el que el estudiante ingresa su código fuente y una consola *REPL* en la que se evalúa dicho código y se obtiene el resultado. De manera simultánea, se visualizan gráficamente las principales representaciones intermedias que se generan y se permite ir avanzando paulatinamente en la ejecución de los procesos internos.

A su vez, previendo un estudiante con más experiencia o materias más avanzadas, se tuvieron en cuenta otros elementos. Se definió un subconjunto del lenguaje *Lisp* que se interpreta, dejando adrede funcionalidades sin implementar para permitir que un estudiante pueda completar o modificar el mismo intérprete y visualizar su funcionamiento con la misma interfaz. También se asumieron criterios de desarrollo y una arquitectura que facilita la modificación puntual de ciertos componentes del software sin necesidad de comprender toda su estructura.

Una decisión clave fue focalizar en los mecanismos de interpretación del código fuente, en particular superar las fases de análisis léxico y gramatical y llegar a abordar el proceso de evaluación, y no necesariamente lograr la generación de código ejecutable o plantear estrategias de optimización.

Por último, otro aspecto que se tuvo en cuenta como horizonte de trabajo, desde la convicción de la importancia estratégica del desarrollo de software de base y del rol de la universidad en el ambiente profesional de sistemas de información, fue dejar abierta la posibilidad de profundizar e ir un poco más allá de los contenidos mínimos que establecen los planes de estudio vigentes.

2. Diagnóstico

El contexto educativo en el que se realizó el estudio es la carrera de Ingeniería en Sistemas de Información de la Facultad Regional Delta de la Universidad Tecnológica Nacional. A la propia experiencia personal de docentes y estudiantes que conforman el equipo de investigación se sumó un relevamiento realizado entre los estudiantes de la carrera como así también a docentes de la misma facultad y de otras casas de estudios.

Dentro del plan de estudios, si bien no hay ninguna asignatura específica, ya sea obligatoria u optativa, sobre creación de lenguajes o que tenga como tema central el

estudio de intérpretes o compiladores, existe una materia en el segundo año de la carrera denominada "Sintaxis y semántica de Lenguajes", que es la que más se aproxima a la temática abordada y donde se gesta la motivación por llevar adelante la presente investigación.

2.1. Intuiciones preliminares

Analizar el funcionamiento de un intérprete o compilador de un lenguaje de programación de alto nivel es una tarea que se puede abordar desde un punto de vista estrictamente teórico, pero como tantos otros aspectos de la informática, la posibilidad de experimentarlo en la práctica, de probar cómo funciona, ayuda a la comprensión de los conceptos.

Una primera aproximación práctica puede realizarse con ejemplos simples en los típicos recursos áulicos de "papel" o "pizarrón". Allí, la ventaja es que el lenguaje a analizar no tiene por qué ser real sino que es posible utilizar una definición de lenguaje hecha especialmente con un fin didáctico, con un conjunto de reglas sintácticas y gramaticales propias, como así también tomar un lenguaje existente y adaptarlo o acotarlo adecuadamente. También permite abordar todo el proceso de compilación o sólo alguna de sus etapas, o hacerlo con diferentes niveles de profundidad.

Un camino diferente, tratándose de código, consiste en utilizar un lenguaje de programación de uso profesional, con su correspondiente compilador o intérprete, y ver en acción los procesos que permiten que finalmente se ejecute un programa escrito en dicho lenguaje, sobre todo al avanzar en casos de mayor complejidad. Lo que sucede, es que no sólo hay que asumir el lenguaje completo tal cual es, sino que generalmente sus compiladores están pensados para realizar eficientemente el proceso y lograr el resultado, pero no dan cuenta de qué manera lo obtuvieron. A su vez, analizar la documentación o su propio código fuente para entender su funcionamiento interno no siempre es posible, no es tarea sencilla y es discutible su sentido educativo.

Una de las intuiciones que fundamenta este trabajo es la posibilidad de hacer converger las ventajas de ambas perspectivas mediante la creación de un intérprete propio de un lenguaje de programación de alto nivel, que ejecute realmente el código fuente como lo hacen los intérpretes profesionales y que a la vez permita focalizar en aquellos aspectos conceptuales que se consideren más significativos y de esta manera sostener un proceso de enseñanza y aprendizaje articulando teoría y práctica.

2.2 Relevamiento

Un relevamiento realizado entre estudiantes y docentes permitió descubrir algunas tendencias en cuanto a dificultades o limitaciones y constatar algunas ideas previas. Los estudiantes consultados son todos de la misma facultad, mientras que entre los docentes hay también quienes provienen de otras instituciones universitarias, en asignaturas con contenidos afines. El instrumento de recolección de datos utilizado fue el mismo para todos, aunque en el caso de los docentes que accedieron a

participar se lo complementó con otras preguntas abiertas que permitieron recavar información caulitativa de interés.

Respecto del proceso de enseñanza y aprendizaje, es contundente la percepción que se trata de un tema complejo y que presenta dificultades. Un primer grupo destaca haber aprendido las etapas de compilación y hacer algunos ejercicios sobre ellas, como por ejemplo graficar autómatas en papel, reconoce que se abordan los elementos y etapas que conforman un lenguaje, pero de manera superficial. Otros señalan que se enseñan los conceptos y partes de un compilador y que se aprende a construir un analizador léxico y un analizador sintáctico mediante un lenguaje de programación. En ese sentido, valoran haber tenido una experiencia de tener que realizar un trabajo práctico donde se implementa alguno de los algoritmos estudiados. Un recurso utilizado es el desarrollo de componentes en relación a la definición de pequeños lenguajes. “Hicimos proyectos interesantes que nos permitieron comprender, por lo menos los primeros pasos, de lo que es crear un lenguaje de programación desde cero”, señala uno de los estudiantes consultados.

En relación a la ubicación de la materias dentro del plan de estudios se evidencia que el grado de profundidad que se puede lograr está limitado en gran parte por la poca experiencia previa en programación que tienen los estudiantes. Esto hace complejo no sólo poder llegar a una implementación práctica, sino que también muestra una falta de contextualización que ayude a interpretar el sentido de los conceptos. Lo cuenta con elocuencia un graduado de la carrera: “vi lo básico de compiladores en un momento de mi vida que no sabía muy bien de qué se trataba programar”.

Consultados puntualmente por las dificultades, las respuestas se pueden organizar mayoritariamente en tres tópicos. Por un lado, en entender el para qué del tema, lo cual guarda relación con la mencionado anteriormente sobre la ubicación de la asignatura en la carrera. Otro aspecto, el más mencionado, se refiere a la complejidad y carácter abstracto de los algoritmos y las estructuras de datos utilizadas; haciendo mención de diversos conceptos como por ejemplo la recursividad, el *backtracking* o los árboles. Un tercer elemento destacable es que aún quienes lograban seguir el funcionamiento de cada componente de manera individual -generalmente los primeros pasos- manifestaban dificultad para comprender el proceso de manera integral. Por último, el alcance predominante de la asignatura incluye el análisis lexicográfico, hay numerosas afirmaciones del estilo "vemos un *parser*", pero que no se avanza a las siguientes etapas, en particular, aquellas que permitan ver un resultado final.

Entre los docentes consultados se manifiesta lo valioso de contar con herramientas pensadas para estudiantes en vez de las que son diseñadas para profesionales, en las que cada docente elige dónde poner el foco o incluso adaptarla a su necesidad.

Resumiendo, se constata un enfoque teórico al que le falta más implementación práctica sobre lenguajes reales y cierta dificultad para ver integralmente el proceso de compilación/interpretación. Recuperando estas impresiones que no pretenden ser exhaustivas o excluyentes de otras realidades, lo que se destaca es que la construcción de lenguajes es un campo en el que hay mucho por hacer y se considera de suma importancia proveer de recursos tecnológicos y pedagógicos para que más docentes,

estudiantes e investigadores den pasos significativos en esta dirección. A su vez, construir una herramienta de software apropiada reafirma la importancia de una interacción fecunda entre teoría y práctica, en particular desde las miradas pedagógicas que interpretan la relación práctica/teoría desde la acción/reflexión [1].

3. Diseño y desarrollo de la herramienta

En sí mismo, el hecho de desarrollar un nuevo intérprete de un lenguaje de alto nivel, por la variedad de componentes de software y conceptos teóricos que combina, puede verse como una experiencia de aprendizaje y de construcción de conocimiento [2]. Pero se buscó dar un paso más allá y lograr que dicho componente se enmarque en una herramienta con sentido educativo.

La herramienta tiene un fin didáctico, no busca optimizar la *performance*, sino que prioriza que sea fácil de usar y ayude a entender mecanismos y conceptos. Se caracteriza por incluir un intérprete de un lenguaje funcional de alto nivel y ofrecer funcionalidades interactivas de visualización de mecanismos, entradas y salidas en los procesos intermedios y brindar la posibilidad de poder modificar, crear y extender los componentes del intérprete y adaptarlo.

Se propone para ser utilizada en asignaturas de la misma carrera universitaria de Sistemas de Información y carreras afines, pero también pueden ser de utilidad en cursos que abordan otro tipo de temáticas relacionadas con el diseño y análisis de lenguajes de programación. No está pensada como primer acercamiento a la programación, sino que puede hacer un mejor aprovechamiento de sus funcionalidades quienes ya tuvieron alguna experiencia en el uso de lenguajes de programación. En otras palabras, se asume como estrategia didáctica general que tiene más sentido profundizar en cómo funciona internamente algo -en este caso un lenguaje de programación- de lo que ya se conoce su utilidad.

De todas maneras, habilita a formas de uso que requieren diferentes niveles de conocimientos previos, por lo que puede utilizarse en espacios académicos que buscan un primer acercamiento a la teoría de lenguajes, como en otros que propongan profundizar en la materia, diseñar intérpretes y compiladores o crear nuevos lenguajes de programación.

Habiendo asumido una metodología de desarrollo incremental, se cuenta con una implementación que ya permite interpretar código y mostrar los resultados, que realiza las tareas principales y se encuentra productiva. En nuevas iteraciones se seguirá completando hasta contar con la herramienta terminada.

3.1. Lenguaje funcional

El lenguaje elegido para analizar y ejecutar es *Lisp*, lenguaje emblemático del paradigma funcional. En particular, se eligió un dialecto denominado *Racket Lisp* [3] que permite realizar las tareas más comunes. En el proceso de selección también se tuvo en cuenta la Construcción de *Lisp* en *Python* [4], por Peter Norvig [5], y un caso

más complejo, “Crafting Interpreters” [6] que toma un lenguaje similar a *Javascript* y muestra todo el camino hasta lograr un intérprete medianamente completo.

Entre las razones principales se encuentra la simplicidad constructiva, sintáctica y semántica, que facilita que los estudiantes y docentes puedan tener una primera experiencia accesible en su camino de aprender sobre compiladores. También, por su alto grado de expresividad. Se trata de una familia de lenguajes con más de 60 años de vigencia: Scheme, CommonLisp, Racket, Closure, entre otros son lenguajes con comunidades vibrantes y una búsqueda por la mejora continua.

Por otra parte, en los últimos años han crecido en el ambiente profesional del desarrollo de software los lenguajes de programación funcionales y muchos de sus conceptos característicos presentes en lenguajes de otros paradigmas. A su vez, en las carreras universitarias de sistemas, no solo de la UTN sino también de otras casas de altos estudios, se enseña la programación funcional. Esto hace que la elección de un lenguaje funcional como Lisp resulte familiar para los estudiantes y no represente una dificultad adicional.

3.2. Diseño preliminar

El diseño de la herramienta consiste en dos componentes principales: el intérprete propiamente dicho, que realiza todos los procesos internos, y una aplicación que proporciona la interfaz de usuario para mostrar y poder seguir de una manera didáctica su funcionamiento.

Las definiciones más importantes se orientaron a establecer los alcances de cada uno de los procesos sucesivos que se aplican sobre el código fuente, la formulación de las representaciones intermedias y la especificación de entradas y salidas de información, de manera de focalizar en los aspectos más significativo para el aprendizaje.

De esta manera, se prioriza el análisis lexicográfico, el análisis sintáctico y el mecanismo de evaluación y como elementos destacadas a mostrar de los resultados intermedios se seleccionó la lista de tokens y el árbol de sintaxis abstracta, siguiendo los criterios clásicos en la materia [7] [8]. Como datos de entrada, se contempla el ingreso del código fuente de la definición de un programa formado por un conjunto de funciones y las consultas que se realizan utilizando dichas funciones. Como datos de salida final, se considera el resultado de la evaluación, en caso que el proceso se complete, o información sobre los errores, si es que se produce alguno.

Por otra parte, en el diseño se dejó de lado el proceso posterior de generar un ejecutable o algún tipo de *bytecode* de más bajo nivel que luego evalúe una máquina virtual, que es un aspecto en el cual difieren los lenguajes actuales de uso industrial. Se trata de una discusión con mucha vigencia en las comunidades que se aglutinan alrededor de cada lenguaje y una decisión clave a la hora de modificar o crear de nuevos lenguajes de programación, y cada vez se encuentran más matices y combinaciones entre los conceptos tradiciones de "compilar" e "interpretar". En todo caso, constituye un posible futuro trabajo pensando en contextos educativos más avanzados o específicos en estas temáticas.

3.3. Uso principal

La utilidad básica consiste en ubicarse como usuario de la herramienta: Se escribe una porción de código que al hacerla evaluar desencadena una serie de procesos, cuyos pasos y estructuras intermedias son mostrados mediante gráficos adecuados, hasta que se obtiene el resultado final. En caso que todo funcione adecuadamente, el énfasis está puesto por un lado en mostrar el paralelismo entre cada porción del código fuente y sus representaciones correspondientes, como también poder visualizar la secuencia interna de evaluación.

Más en detalle, se comienza con el ingreso del código fuente de un programa sencillo en *Lisp*. Por ejemplo, en la figura 1 se detalla cómo se define una función.

```
(define (siguiente valor) (+ valor 1))
```

Fig 1. Definición de una función

Los *tokens* que genera el análisis lexicográfico se visualizan como una lista, en la que se refleja si se trata de identificadores, de palabras reservadas, literales o símbolos. La figura 2 muestra la lista de *tokens* correspondiente al código de la figura anterior.

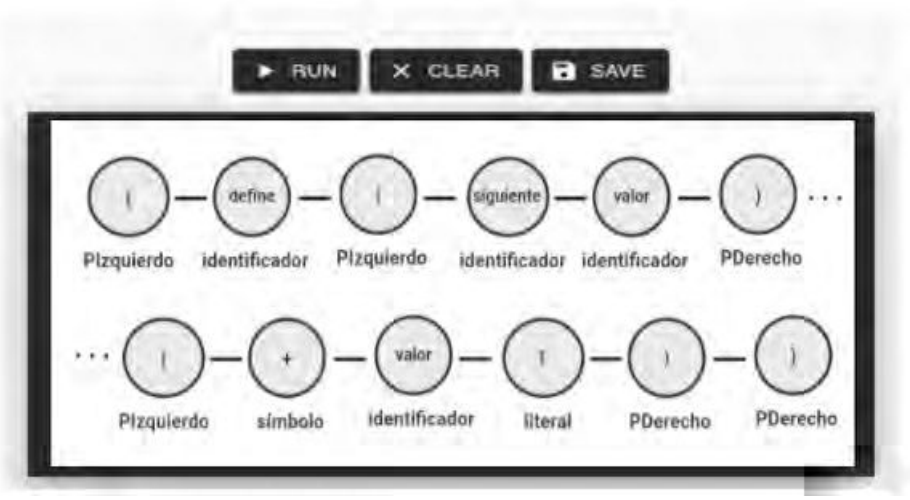

Fig 2: Lista de *tokens*

Luego, el análisis sintáctico “parsea” los *tokens* y construye el árbol de sintaxis abstracta, mejor conocido como AST, por su sigla en idioma inglés. En la figura 3 se muestra el árbol correspondiente al ejemplo anterior. Puede observarse que ya no

están presentes elementos como los paréntesis que son innecesarios en esta etapa y los diferentes tipos de nodos con sus respectivos colores.

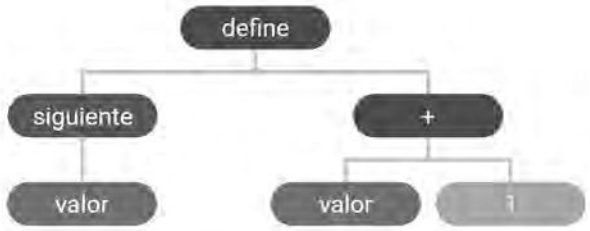

Fig 3: *Abstract Syntax Tree*

La instancia de evaluación implica analizar la expresión ingresada en la consola *REPL* e incorporarla al árbol ya construido, para luego recorrerlo realizando las sustituciones correspondientes, de manera de generar un resultado final que se vuelve a mostrar en dicha consola (Ver figura 4).

Fig 4: Consola REPL con el resultado de la evaluación.

3.4. Forma de uso alternativa

Una utilidad avanzada, que sin dudas requiere de un acompañamiento docente más intenso y un perfil de estudiantes con mayor experiencia, consiste en modificar el componente del intérprete mismo, y utilizando la misma interfaz y demás elementos

de la aplicación poder ver cómo se comporta diferente. Para ello, además de haber cuidado criterios de expresividad en el código y una adecuada modularización para que sea más sencillo de modificar, se dejaron adrede "huecos" en la formulación del lenguaje para habilitar a la realización de consigna de trabajo acotadas y precisas (por ejemplo, completar un tipo de dato faltante, agregar operaciones sobre un tipo existente, modificar una palabra reservada, etc.).

3.5. Decisiones técnicas

Se decidió utilizar *Rust* [9] para la construcción del intérprete propiamente dicho, teniendo en cuenta que es un lenguaje compilado, estáticamente tipado, moderno y con un enfoque en corrección que se destaca en el panorama actual de la industria de desarrollo de software. A la vez, comparte ciertos principios de funcionamiento con *Lisp* lo que le da mayor coherencia y robustez al desarrollo.

El código escrito en *Rust* se compila a un paquete de *WebAssembly*, que es un lenguaje de código binario que se puede ejecutar en el navegador. *React* consume el paquete de *WebAssembly*, lo que permite acceder a las funciones que expone el intérprete en un navegador.

4. Resultados y futuros trabajos

En primer lugar, en cuanto a la construcción de la herramienta como tal, los resultados alcanzados hasta ahora son satisfactorios, teniendo en cuenta que las funcionalidades desarrolladas funcionan de acuerdo a lo previsto y lo que se encuentra pendiente no impide el funcionamiento del resto.

El subconjunto de definiciones contempladas válidas de *Lisp* se evalúa satisfactoriamente, se obtiene el resultado y se visualizan los gráficos cuando el código es correcto. Las expresiones que producen un error en el *Lisp* original también dan un error en esta implementación y despliegan una leyenda similar, aunque no se exprese aún gráficamente.

En forma experimental, simulando un posible uso avanzado por parte de estudiantes y tomando como punto de partida una versión ya bastante avanzada de la herramienta, se hizo el ejercicio de puntualmente agregar un nuevo tipo de dato e implementar ciertas operaciones nativas sin alterar el resto del código. Reconociendo la distancia entre quien es miembro del equipo de desarrollo y quien no, aún tratándose potencialmente en ambos casos de estudiantes avanzados, se trata de un resultado provisorio pero prometedor.

Recuperando lo dicho anteriormente de que se trata de un trabajo en progreso, en cuanto al desarrollo mismo queda pendiente una mejor y más precisa forma de comunicar y mostrar los errores.

En la versión actual se informa que se produjo un error, indicando el motivo mediante una descripción textual e interrumpiendo el proceso de interpretación. (Ver figura 5). Entendiendo la importancia que tiene para un estudiante enfrentarse a un

error como parte de su proceso de aprendizaje, lo que se espera es poder representarlo gráficamente integrando la visualización de dicho error en los diagramas que se despliegan cuando el proceso termina correctamente. A su vez, dependiendo del motivo, ubicación o gravedad del error, se busca habilitar total o parcialmente la continuidad del proceso o habilitar la evaluación de las porciones del código fuente que no presentan errores.

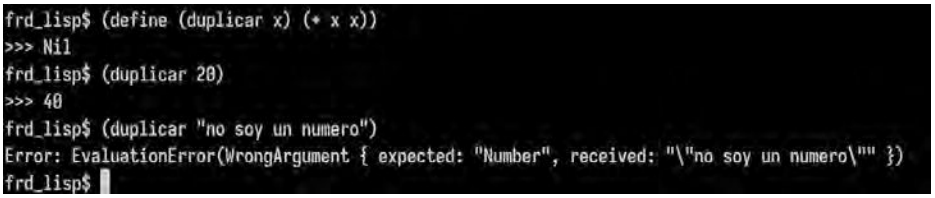

Fig 5: Visualización de un error de evaluación

Aún no se cuenta con los resultados reales de ver el impacto del uso de herramienta en estudiantes, lo que dará elementos más confiables acerca del logro de los objetivos planteados. Previendo esa instancia, el relevamiento realizado inicialmente establece un punto de referencia a la hora de realizar un nuevo estudio con estudiantes que hayan utilizado la presente herramienta y resultará oportuno utilizar como base el mismo instrumento de recolección de datos, sumándole alguna pregunta adicional. Cabe reconocer que los tiempos del recorrido académico de los estudiantes y de las decisiones de los docentes no son necesariamente los mismos que los que se dan en la dinámica de la investigación, por lo que la obtención de resultados requerirá como mínimo de un año. Asimismo, teniendo en cuenta que el período tomado para el relevamiento inicial abarca varios años de cursada de la materia para tener una muestra más sólida, sería consistente contemplar también más de una cursada con esta nueva herramienta.

Referencias

1. Freire, P. (1970) Pedagogía del Oprimido. Ed. Tierra Nueva. Montevideo
2. Moreno-Seco, Forcada. (2001). Learning compiler design as a research activity. Departament de Llenguatges i Sistemes Informatics, Universitat d'Alacant
3. Racket Lisp. 2019 Sitio web https://racket-lang.org
4. Lispy. 2019 Sitio web http://norvig.com/lispy.html
5. Norvig, Peter. 2019 Sitio web https://en.wikipedia.org/wiki/Peter_Norvig
6. Bob Nystrom.Crafting Interpreters. 2019 Sitio web http://craftinginterpreters.com
7. Aho, Lam, Sethi, Ullman. (1986). Compilers: Principles, Techniques, and Tools. Addison Wesley.
8. Cooper, Torczon. (2011). Engineering a Compiler. Morgan Kaufmann
9. Rust Programming Language. 2019 Sitio web https://www.rust-lang.org