Facultade de Informática
UNIVERSIDADE DA CORUÑA

TRABALLO FIN DE GRAO
GRAO EN ENXEÑARÍA INFORMÁTICA
MENCIÓN EN COMPUTACIÓN

Lenguaje de programación funcional ejecutable sobre una máquina virtual propia

Estudante: Eduardo Javier Ferro Sastre

Dirección: Laura Milagros Castro Souto

A Coruña, septiembre de 2022.

Dedicado a mi madre.

Agradecimientos

Quiero agradecer este trabajo a mi madre, mi pareja, Archi y mis amigos que me han apoyado durante estos años de trabajo.

Resumen

En el presente trabajo se lleva acabo el diseño de un lenguaje de programación funcional junto a la implementación de un compilador para el propio lenguaje, a su vez, se implementa una máquina virtual capaz de ejecutar las instrucciones resultantes de la compilación. Con este propósito se muestran los distintos algoritmos para analizar y detectar el lenguaje, así como los puntos a favor y en contra de cada uno. También se incluye una explicación sobre las bases del funcionamiento de una máquina virtual y una descripción detallada de la estructura y procesos utilizados en la implementación de esta. Como última fase del proyecto, se añade una capa de compatibilidad del lenguaje con otras máquina virtuales ya existentes y se estudian sus diferencias con la máquina implementada para el lenguaje.

Abstract

In the present work, the design of a functional programming language is carried out together with the implementation of a compiler for the language itself, and a virtual machine capable of executing the resulting instructions of the compilation. With this purpose in mind, the different algorithms to analyze and detect language are shown, as well as the points for and against each one. There is also included an explanation of the basics of a virtual machine and a detailed description of the structure and processes used in its implementation. As the last phase of the project, a layer of language compatibility with other existing virtual machines is added, and the differences with the machine implemented for this language are studied.

Palabras clave:

- Máquina Virtual
- Lenguaje de Programación
- Paradigma Funcional
- Compilador
- Bytecode

Keywords:

- Virtual Machine
- Programming Language
- Functional Programming (FP)
- Compilator
- Bytecode

Índice general

Índice de figuras

Índice de tablas

Capítulo 1

Introducción

Los lenguajes de programación, la herramienta con la que nos comunicamos con un **dispositivo electrónico**, son una parte primordial del desarrollo. La sociedad actual depende en gran medida de las facilidades que nos ofrecen estos dispositivos por lo que es importante hacer esta comunicación lo más sencilla posible. Para dar forma a estos lenguajes existen distintos **paradigmas**, modelos de como se forman y dan uso a los mismos con reglas especificas. En las últimas décadas, los lenguajes propios del paradigma de la programación funcional han comenzado a ganar importancia tanto por sí solos como en lenguajes de otros paradigmas de la programación, añadiendo a estos características típicamente funcionales como por ejemplo la coincidencia de patrones (conocida en inglés como pattern-matching).

En este trabajo se pretende ahondar en el **diseño e implantación de un lenguaje funcional** que se compile a instrucciones capaces de ejecutarse sobre una **máquina virtual propia**. Dicho lenguaje busca ser una **opción atractiva y sencilla** que, aparte de contener los aspectos propios de la **programación funcional**, incluya también algunos aspectos propios del paradigma de la **programación orientado a objetos (POO)**, conocida en inglés como *Object-Oriented Programming (OOP)*. A su vez, también se procura ahondar en cómo funcionan los lenguajes de programación y las máquinas virtuales como entornos de ejecución.

1.1 Objetivos

Una vez establecidas las ideas principales que dieron lugar a este proyecto, en esta sección introduciremos cuales serán los objetivos del mismo:

- Desarrollo de un **lenguaje de programación funcional** con las siguientes características:
  - **Sintaxis** donde se priorice la **simplicidad**.

  - **Capacidad de ejecutar el código de manera paralela** y tener **comunicación entre distintos hilos de mensajes**.
  - Gestionar la **existencia de clases** y la **herencia**, con el objetivo de realizar factorización y reutilización de código.

- **Implementación de un compilador** que transforme el código fuente en el lenguaje deseado en un **código intermedio, *bytecode,* de instrucciones propias**, con la posibilidad de **exportarlo** a otros **códigos intermedios compatibles** con la **máquina virtual de Java (JVM)** o con la **máquina virtual de Erlang (BEAM)**.
- **Implementación de una máquina virtual** para la ejecución del *bytecode* de instrucciones propias.

1.2 Organización de la memoria

Con el fin de facilitar la lectura y el seguimiento del presente documento, a continuación se proporcionará a la lector/a un listado y una breve descripción de los capítulos que la conforman.

- **Capítulo 1: Introducción**. Este capítulo incluye una breve introducción al proyecto junto a la definición de los objetivos que se pretenden alcanzar en el mismo.
- **Capítulo 2: Fundamentos tecnológicos.** Este capítulo se centrará en familiarizar a la lector/a con el dominio técnico: Algoritmos de análisis sintácticos para compilación del lenguaje, máquinas virtuales, etc.
- **Capítulo 3: Metodología.** Este capítulo se encargará de describir la planificación del proyecto, incluyendo la metodología escogida, el desglose de las tareas, los recursos utilizados (tanto hardware como software) y los costes asociados a los mismos.
- **Capítulo 4: Implementación.** En este capítulo se hablará de como se ha llevado a cabo la implementación del lenguaje de programación, de la propia máquina virtual y del compilador.
- **Capítulo 5: Conclusiones.** Después de haber realizado la implementación, en este capítulo se comentarán cuales han sido las conclusiones extraídas, evaluando los objetivos alcanzados, realizando un seguimiento del proyecto y exponiendo sus posibles líneas futuras.

Capítulo 2

Fundamentos Tecnológicos

Este capítulo pretende familiarizar a la lector/a con el dominio, brindándole los recursos necesarios para entender los conceptos **teóricos** empleados en el desarrollo del proyecto y el entendimiento de la memoria. A fin de lograrlo, este capítulo contendrá, entre otros conceptos, las técnicas más empleadas en el **estado del arte** a la hora de crear un lenguaje, una máquina virtual o un compilador: centrándose la primera parte en el análisis **léxico** y **sintáctico** del código, sus algoritmos y herramientas actuales; y la segunda parte, en la implementación de **máquinas virtuales**, su funcionamiento y los distintos tipos de aproximaciones a la ejecución.

Tanto las máquinas virtuales (*VM*, por sus siglas en inglés) como los ordenadores actuales están basados en los fundamentos sobre los autómatas y su consecuente desarrollo futuro. Este desarrollo se divide principalmente en dos vertientes en las que ahondaremos en futuras secciones: máquinas puramente **interpretadas** y máquinas **pseudo-compiladas**. Con el fin de que estos dos tipos de máquinas puedan ejecutar e interpretar una serie de líneas de código de un lenguaje origen, es necesario emplear un compilador que analice y comprenda qué expresa dicho código, el cual debe mantener un formato y unas reglas (tanto léxicas como sintácticas) muy firmes que nos ayuden a detectar la intención del mismo con la menor ambigüedad posible.

Los **compiladores** son **programas desarrollados para analizar** un conjunto de líneas de **código escritas en un lenguaje de programación**, conocido como Código Fuente, y transformarlas en otro lenguaje distinto, al que llamaremos lenguaje de destino. Dependiendo de su uso final, el lenguaje de destino puede ser desde un **lenguaje máquina**, código binario interpretable por los circuitos programables más básicos (CPU), a un lenguaje de más alto nivel como por ejemplo transformación a formato PDF o, hasta a un lenguaje entendible por una máquina virtual, conocido como **código intermedio** (Byte-Code), proceso recogido en la figura 2.1. La utilidad de los compiladores es basta dado que su función principal es el de comprender la estructura de un texto y transformarlo en otro distinto.

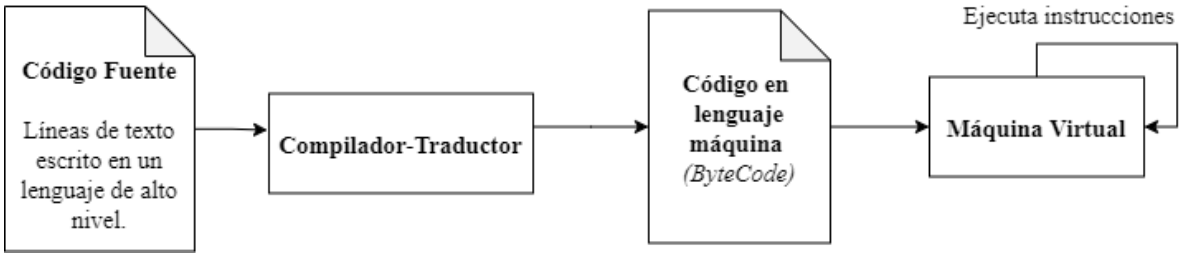


Figura 2.1: Esquema de flujo desde el código fuente hasta su ejecución.

Los compiladores se dividen principalmente en **dos tipos** según su finalidad: **Compiladores-Traductores** (que a partir de ahora llamaremos únicamente Compiladores), encargados de únicamente transformar el código de entrada en un código de salida, e **Intérpretes**, que además de su proceso de traducción realizan una interpretación del texto, a menudo acompañados de una acción concreta a ejecutar. Otros sistemas más complejos, como el utilizado en el lenguaje de programación JAVA, constan de una mezcla de ambos, pero como veremos más adelante tan solo son la unión de un Compilador-Traductor con un Intérprete en dos procesos separados.

Para llevar a cabo la tarea de traducción, ambos tipos de compiladores deben **analizar la estructura del texto**, por lo que dicha explicación será común a ambos. Este proceso de **análisis del código fuente** se divide en dos partes: **análisis léxico**, que se realiza mediante un Lexer, en el que reconoceremos conjuntos de símbolos transformándolos en una lista de símbolos más simples (también conocidos como Tokens), y un **análisis sintáctico**, que se realiza mediante un Parser, en el que se tendrá en cuenta la posición y contexto de cada token con sus predecesores y sucesores además de su significado. Tanto el funcionamiento del Lexer como del Parser se explicarán a continuación en las siguientes secciones.

2.1 Lexer

Los analizadores léxicos, también llamados **Lexers** son **procesos encargados del análisis léxico del código**. Estos procesos se basan en, a través del reconocimiento de patrones de símbolos, acabar detectando elementos complejos. Ejemplo de ello, es el poder detectar Identificadores, nombres utilizados habitualmente para las variables, caracterizados entre otras reglas por no estar incluidas en las palabras clave, como podrían ser“if” o “while”.

Los **analizadores léxicos** habitualmente utilizan **máquinas capaces de detectar expresión regular/es**, secuencias de caracteres que conforman un patrón. Un ejemplo de expresión regular sería la empleada para detectar identificadores, que se caracteriza por aceptar un conjunto de caracteres alfanuméricos que pueden incluir una o más “_”. De esta manera, gracias a las expresiones regulares se pueden generalizar los distintos tipos de Tokens a detectar.

Las máquinas que se utilizan para detectar estas expresiones regulares se llaman **máquinas de estado finito**. Con este tipo de máquinas se pretende realizar un número finito de pasos de un estado a otro según las entradas que reciban. Los diagramas que muestran el funcionamiento de estas máquinas utilizan círculos para representar los estados, siendo un círculo doble si lo que se está a representar es un estado final, y flechas apuntando al estado destino junto a la entrada necesaria para realizar el cambio de estado.

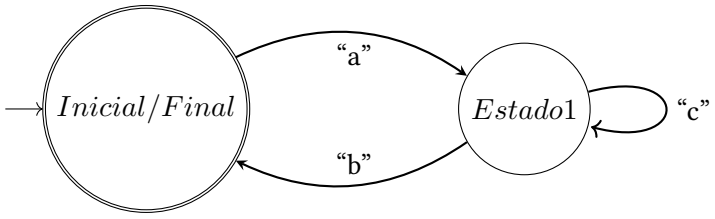


Figura 2.2: Ejemplo de diagrama de estado.

Centrándonos en el ejemplo de la figura 2.2, podemos ver como en este ejemplo de máquina de estado finito aceptaría la expresión regular "(ac*b)*". Dicha máquina contiene dos estados, el estado $Inicial/Final$ y el estado $Estado1$. Desde el estado $Inicial/Final$ solo se puede recibir el carácter "a", desde el Estado 1 se puede recibir tanto el carácter "c" cero o más veces (quedando en el propio $Estado1$), como el carácter "b" una única vez (volviendo al estado $Inicial/Final$).

A la hora de detectar estas expresiones en lenguajes naturales o en lenguajes de programación, se suele utilizar el espacio en blanco como separador de palabras, dado que puede ser complicado definir de otra manera que es una palabra o Token por si sola. El separador sirve como ayuda para la **detección de palabras**, pero hay casos muy habituales dentro de los lenguajes de programación en los que sin la posibilidad de espacios debemos separar dos Tokens. Este es el caso de la llamada a una función, que en lenguajes como JAVA y C++ se escribe empleando dos Tokens, un identificador seguido de una apertura de paréntesis, "identificador(". En estos casos no siempre hay espacio que los separe por lo que se puede utilizar un mecanismo de *look-ahead* o un mecanismo de *backtracking* (explicados con más detalle en la Sección 2.2) en el que se acepte el posible token solo una vez se compruebe el siguiente carácter.

Estos métodos también ayudan a resolver el problema llamado "*maximal munch*" [1] que surge de la necesidad de consumir o **eliminar de la entrada el número máximo de caracteres** posibles que correspondan a la expresión. De no solventar este problema, el analizador léxico podría devolver el primer conjunto que correspondiera con la unidad mínima aceptada por la máquina incluida en la expresión, por ejemplo en el caso de la figura 2.2 utilizando la entrada "accbacb" podría devolver el Token "accb" sin llegar a comprobar toda la cadena, la cual también es aceptable por la máquina. Por lo tanto, en vez de aceptar como uno solo "accbacb", encontraría en esa cadena dos Tokens: "accb" y "acb".

Estas máquinas que se emplean como lexer se pueden crear a mano o con herramientas como los generadores de lexers. Entre los generadores de lexers más utilizados están "lex"[2] y "flex"[3], siendo el segundo una reimplementación del primero. Estos son capaces de detectar Tokens en base a expresiones regulares y comunicarse con generadores de parsers para trabajar en conjunto.

2.2 Parsers

Los **analizadores sintácticos** o *parsers* más populares se dividen en categorías según su algoritmo y su forma de analizar los tokens. Estos analizadores toman una serie de tokens sucesivos que representan los elementos del código original y los analizan a fin de construir un árbol sintáctico empleando las reglas de la **gramática** del lenguaje. Más tarde ese mismo árbol se recorre para poder realizar un proceso de reconocimiento en el que se pueden comprobar los tipos correspondientes a funciones y variables a través de la **estructura de datos**.

2.2.1 Gramáticas

Para poder generar un árbol sintáctico correcto de los tokens es necesario conocer la **Gramática del lenguaje**, esto es la serie de reglas que componen como se forma este lenguaje y como se enlaza entre sí. Estas reglas se llaman **Reglas de Producción**, y se conforman por una parte izquierda y una parte derecha con la forma: $S \rightarrow A$, siendo S la parte izquierda o cabeza de la regla y A la parte derecha o cuerpo.

Los elementos que se encuentran a cada parte de la regla pueden ser tokens **Terminales** o **No-Terminales**. Los tokens **Terminales** son aquellos que **no están conformados por otros tokens**, y por lo tanto, no aparecen nunca en la parte izquierda. Sin embargo, los **No-Terminales** son aquellos que existen como resultado de la **composición de otros Tokens** (tanto Terminales como No-Terminales).

Asimismo, por ejemplo si existieran unas reglas de producción tales que:

- $S \rightarrow Abbc$
- $A \rightarrow zz$

Siendo los elementos b, c y z Terminales, ya que no aparecen en la parte izquierda de ninguna regla, al contrario que los elementos S y A, los cuales son No-Terminales.

Por costumbre, y a lo largo de este documento, los elementos Terminales se escriben en minúscula o como letras de alfabeto griego y los No-Terminales en mayúscula. El primer elemento, el que compone todo, se suele llamar S (por su inicial en ingles de la palabra "*Start*") y el elemento vacío que representa la entrada de ningún token, se representa con ϵ o λ. En este documento usaremos ϵ por simplicidad y por ser la inicial del termino vacío en inglés,"*Empty*".

Según el conjunto de reglas de producción que forman una gramática se pueden distinguir dos tipos de gramáticas: la **gramática ambigua** y la **gramática no-ambigua**. La **gramática ambigua** es la que **para una misma cadena se puede generar diferentes árboles sintácticos** y, por el contrario, la **gramática no-ambigua** es aquella que **para una cadena solo existe un árbol sintáctico posible**.

Las gramáticas también se pueden dividir empleando la **jerarquía de Chomsky**[4], quien es un lingüista, filósofo y científico cognitivo especializado en esta materia. Esta jerarquía se creó en base a cuanta flexibilidad tenían estas gramáticas para expresar sus correspondientes lenguajes. Los tipos de gramáticas avanzan según lo planteado en la figura 2.3: desde el tipo cero, Gramáticas No-Restringidas, que abarcan todos los lenguajes aceptados por una máquina de Turing, hasta las gramáticas de tipo tres, Gramáticas Regulares con una estructura muy específica. A continuación, cada uno de estos tipos se explicarán más en detalle.

Tipo Cero: Gramáticas No-Restringidas

Las gramáticas del tipo cero abarcan todas aquellas reconocibles por una máquina de Turing[5] y, por lo tanto, todas aquellas gramáticas de los tipos sucesivos (Tipo uno, dos y tres). Estas gramáticas son las más flexibles pero también las más difíciles de detectar por las posibilidades que reflejan.

Tipo Uno: Gramáticas Sensibles al Contexto

Las gramáticas de tipo uno generan lenguajes sensibles al contexto, estos lenguajes son aquellos en los que la aceptación de algunos token cambia dependiendo de lo que les rodea, esto es el **contexto**.

También se definen como gramáticas sensibles al contexto aquellas que sus reglas cumplan con la forma siguiente:

- $\alpha \mathrm{A} \beta \rightarrow \alpha\ \gamma\ \beta$
- A siendo un No-Terminal.
- $\alpha\ \gamma\ \beta$ siendo un Terminal o un No-Terminal.
- $\alpha\ \beta$ pueden estar vacíos.
- γ no puede estar vacío.
- La regla $S \rightarrow \epsilon$ está permitida si S no aparece en ninguna parte derecha de ninguna regla.

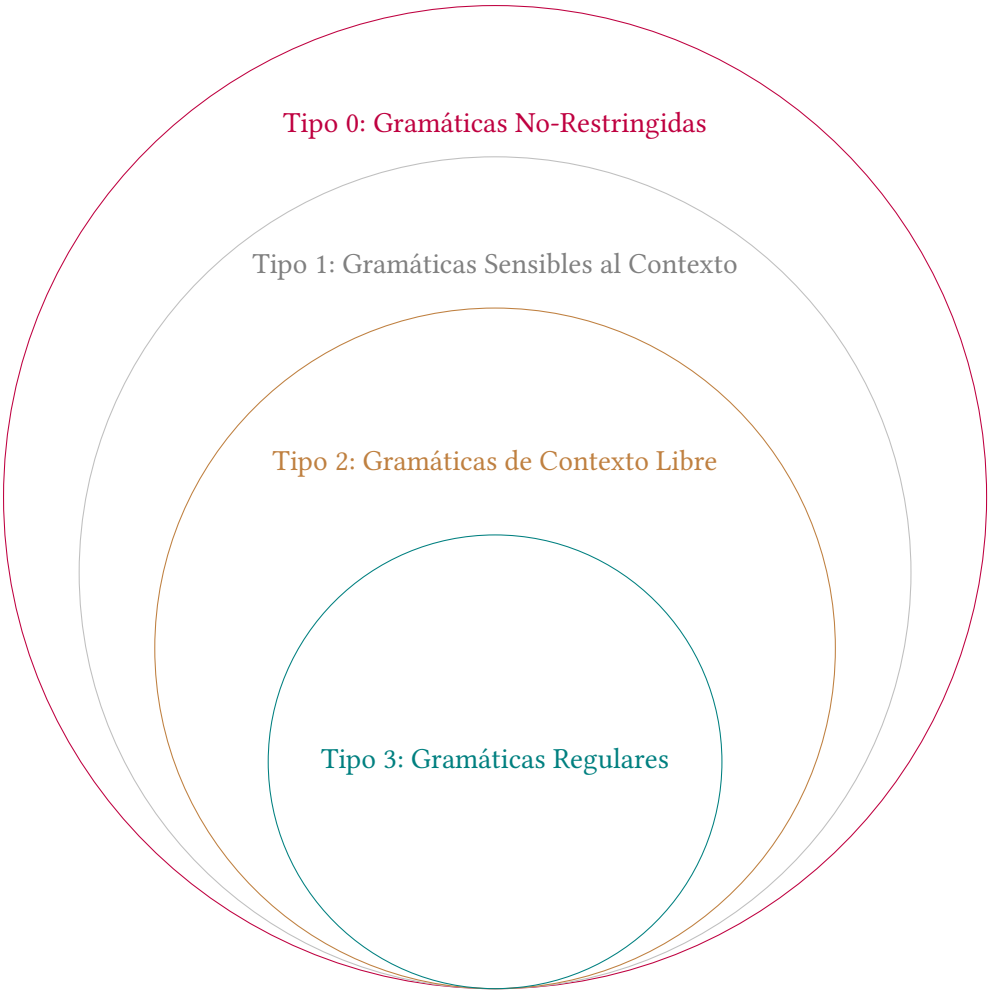


Figura 2.3: Tipos de Gramáticas.

Tipo Dos (I): Gramáticas de Contexto Libre

Las gramáticas de tipo dos generan **lenguajes de contexto libre**, esto quiere decir que al contrario que las anteriores **no dependen de sus alrededores para definir la validez de un token en una posición en concreto**, dejando menos opciones de lenguajes a generar.

Estos lenguajes están definidos por la forma:

- $A \rightarrow \alpha$
- A siendo un No-Terminal.
- α siendo una cadena de Terminales y/o No-Terminales.

Los lenguajes de programación habitualmente se representan con un subconjunto de Gramáticas Libres de Contexto llamado **Gramáticas Libres de Contexto Deterministas**. A su vez, algunas veces se utiliza un subconjunto de esas mismas gramáticas para ser analizado de forma más fácil o rápida por un algoritmo de análisis como LL, que explicaremos más adelante.

Tipo Dos (II): Gramáticas de Contexto Libre Deterministas

Los lenguajes formados por **Gramáticas de Contexto Libre Deterministas** (*DCFL*, por sus siglas en inglés) son aquellos que pueden ser reconocidos por una **máquina de Turing determinista en un tiempo polinomial y con espacio de** $O(log^2 n)$.

Tipo Tres: Gramáticas Regulares

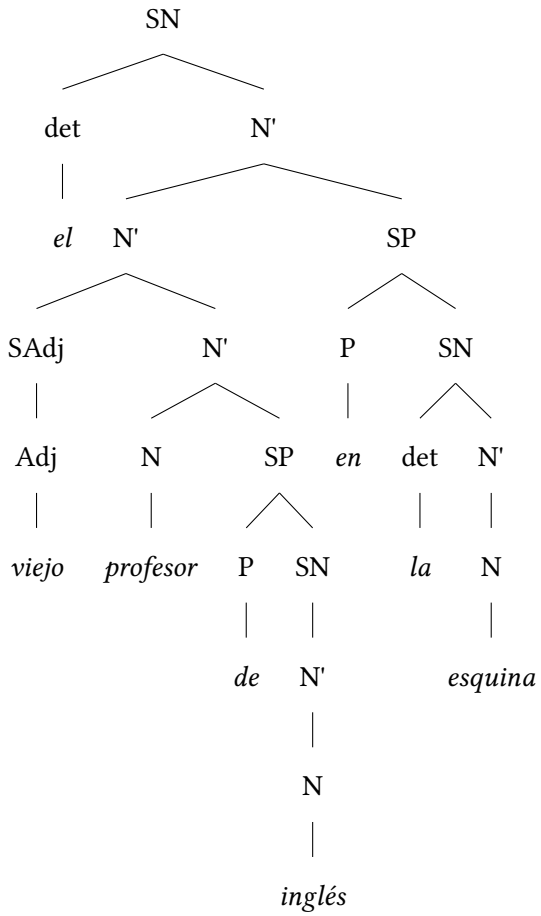
Las **Gramáticas Regulares** generan **lenguajes regulares** que se pueden definir con **expresiones regulares** y pueden ser reconocidos por un **autómata finito**.

Estos lenguajes están definidos por la forma:

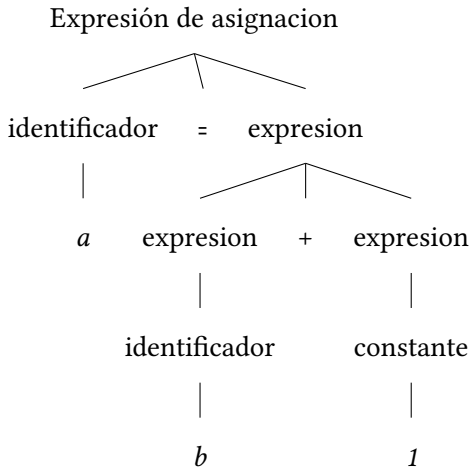
- $A \rightarrow \alpha B$ (derecha regular).
- $A \rightarrow B\alpha$ (izquierda regular).
- Sin mezclar reglas de derecha regular e izquierda regular.
- α siendo un Terminal.
- B siendo un No-Terminal.
- $S \rightarrow \epsilon$ está permitida si S no aparece en ninguna parte derecha de ninguna regla.

2.2.2 Estructuras de datos para análisis

Los *parsers* utilizan las reglas de las gramáticas y sus características para detectar y construir estructuras que representen el código. La **estructura más utilizada para almacenar y recorrer estos datos** son los **árboles sintácticos**. Como se puede ver en la figura 2.4a, cuando analizamos sintácticamente una oración en lenguaje natural podemos construir un árbol sintáctico, de igual manera, el código escrito en un lenguaje de programación sigue un esquema similar, como podemos observar en la figura 2.4b.



(a) Árbol con análisis sintáctico en español.



(b) Árbol con análisis sintáctico en programación.

Figura 2.4: Árbol con análisis.

La construcción de estos árboles se puede realizar de distintas maneras, mayoritariamente divididas en dos grupos: *bottom-up* y *top-down*. En el primero, como bien indica su nombre se construye el árbol desde abajo y subiendo, se parten de los tokens terminales para acabar

llegando al token inicial y, en el segundo, se deriva el árbol desde la cima, desde el token inicial S hasta los tokens terminales.

2.2.3 Top-down

Los algoritmos de *Top-Down*[6] parten de un token No-Terminal inicial S y van derivando a una serie de posibles árboles hasta encontrar el adecuado a la entrada. De esta manera se van recorriendo todos los posibles árboles hasta formar la estructura árbol final, que representa una cadena desde los tokens de entrada y una serie de aplicación de reglas.

Estos algoritmos son beneficiosos en la simpleza que acarrea su implementación, pero poco eficientes al utilizarlos sobre gramáticas ambiguas, o con reglas de producción que generan cadenas similares como por ejemplo $B \rightarrow \beta|\beta\gamma$. Esto los vuelve poco viables para los lenguajes de programación o ámbitos en que las gramáticas pueden ser muy complejas.

Para una gramática con las reglas:

1. - $S \rightarrow \alpha BC$
2. - $B \rightarrow \beta|\beta\gamma$
3. - $C \rightarrow \delta\zeta$

Al recibir la cadena de entrada "$\alpha\ \beta\ \gamma\ \delta\ \zeta$" se expandirá primero la regla inicial "$S \rightarrow \alpha BC$". Después la correspondiente con el símbolo B, pero al existir dos posibilidades (β o $\beta\gamma$) deberá esperar a la siguiente entrada (γ) para descartar la primera derivación de la regla dos y aceptar la segunda. Finalmente, se expandirá la tercera, con lo que ya habremos obtenido toda la cadena.

Algunos algoritmos utilizan una **lectura anticipada** para optimizar la generación del árbol, de esta manera no será necesario explorar todas las posibilidades tan exhaustivamente. Los algoritmos que utilizan la lectura anticipada (también llamada en inglés "*look-ahead*") se simbolizan con el valor de tokens anticipados entre paréntesis. Por ejemplo, el algoritmo $LL(1)$ que explicaremos a continuación, tiene solo una posición de lectura anticipada.

LL

Uno de los algoritmos más conocidos dentro de los *Top-down* es el LL[7] (por sus siglas en inglés *Left-to-right Leftmost derivation*). Este algoritmo se basa en la creación del árbol por medio de derivar los símbolos no terminales de izquierda a derecha, escogiendo primero el símbolo más a la izquierda. Veamos un ejemplo a continuación:

Dada una gramática con las reglas:

- $S \rightarrow S + S$

- $S \rightarrow 1$
- $S \rightarrow \alpha$

Si quisiésemos generar la cadena de entrada: "$1 + 1 + \alpha$", partiríamos del no terminal inicial S y, a continuación, se derivaría el símbolo inicial S con la primera regla "$S \rightarrow S + S$", obteniendo:

- $S + S$

Después, se derivaría el símbolo "S" izquierdo con la segunda regla "$S \rightarrow 1$", obteniendo:

- 1 + S

En este punto si se utilizara la segunda regla tendría que retroceder al no coincidir con la entrada, y probar con la primera regla.

- $1 + S + S$

A continuación, derivando el no terminal de más a la izquierda, se utilizaría la segunda regla "$S \rightarrow 1$".

- $1 + 1 + S$

Por último, se derivaría el único símbolo restante No-Terminal con la tercera regla "$S \rightarrow \alpha$".

- $1 + 1 + \alpha$

Como se aprecia en el ejemplo, la derivación de los elementos comienza de izquierda a derecha, derivando el No-Terminal S en "1". Si por el contrario este algoritmo se resolviera de derecha a izquierda, el proceso sería similar pero el primer No-terminal a resolver, después del inicial, sería el de la derecha, substituyéndolo por α.

Cabe destacar que la gramática utilizada en este ejemplo es ambigua, por lo que se podría conseguir otro árbol que concuerde con la entrada y sigue sus mismas reglas. En este caso, como se aprecia en la figura 2.5, a la hora de derivar la estructura desde el paso número dos "$S + S$" a "$1 + S$" utilizando la segunda regla, es posible en su lugar derivar a "$S + S + S$" utilizando la primera regla y ambos árboles son válidos.

En la sección 2.3, donde se habla del estado del arte y de las herramientas actuales, ahondaremos en como los generadores de *parsers* deciden cual de las dos opciones es la adecuada.

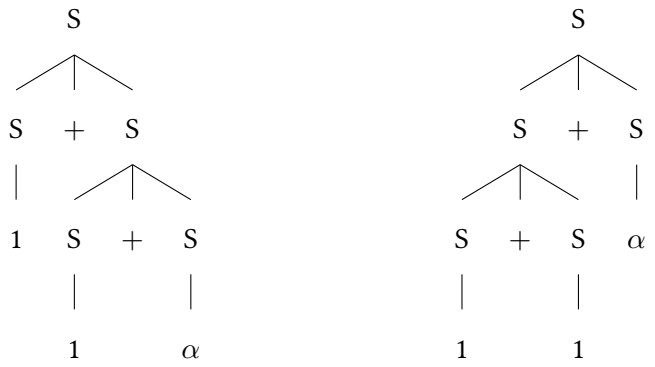


Figura 2.5: Ejemplo de gramática que puede formar árboles ambiguos.

2.2.4 Bottom-up

Los algoritmos del grupo *Bottom-up*[8] crean su árbol a través de reducir los tokens de la cadena a uno más general, subiendo así desde abajo a la vez que se forma la estructura del árbol hasta llegar al token inicial. Estos algoritmos usualmente consideran dos opciones principales, *Shift* (avance o cambio) y *Reduce* (reducción). *Shift* consiste en avanzar uno a uno en los tokens de entradas, mientras que, *Reduce*, por otro lado, consume un conjunto de los últimos tokens para reducirlos a uno nuevo más general.

Estos algoritmos se benefician de gramáticas construidas sobre un gran número de reglas con principios de regla similares pero finales distintos, eliminando la necesidad de uso del mecanismo de *backtracking* debido a seguir las suposiciones iniciales. Los algoritmos más conocidos parten de un algoritmo común que es el LR, que como veremos a continuación, utilizan una tabla para decidir cual de las dos opciones anteriormente mencionadas se debe utilizar.

LR

El **algoritmo de *parsing*** LR**[9]** (por sus siglas en inglés: *Left-to-right Rightmost derivation in reverse*), sigue el mismo orden de izquierda a derecha que el algoritmo LL pero la reducción se realiza comenzando por la opción más a la derecha. El algoritmo LR, al igual que LL, puede utilizarse con una lectura anticipada.

Utilizando las mismas reglas que en el ejemplo anterior para LL, veremos como avanzaría el análisis de la misma cadena de tokens "$1 + 1 + \alpha$" pero, esta vez aplicando un algoritmo LR.

Al igual que en ejemplo de la página 11, la gramática está compuesta por las reglas:

- $S \to S + S$
- $S \to 1$

- $S \rightarrow \alpha$

Y la cadena de entrada sigue siendo la ya mencionada:"$1 + 1 + \alpha$".
En este algoritmo se analizaría y consumiría primero el primer token Terminal "1".

- 1

Después se utilizaría la segunda regla, reduciendo el token Terminal 1 a uno No-Terminal S.

- S

Al no poderse reducir más se avanza, empleando el *Shift*, una posición.

- $S+$

Como $S+$ no se puede reducir utilizando ninguna regla conocida, se avanza otra posición de la cadena.

- $S + 1$

Al ahora sí poder reducirse y consumirse el último token, se aplica la segunda regla.

- $S + S$

El resultado de esto, sí es reducible otra vez, empleando la primera regla.

- S

Al no ser reducible y no acabarse la cadena de entrada, se avanza una posición.

- $S+$

De nuevo, no es reducible por lo que se avanza otra posición al token restante.

- $S + \alpha$

Este token α sí es reducible por la tercera regla de producción.

- $S + S$

Por último, esto es reducible utilizando la primera regla y llegando así la raíz del árbol.

- S

Debido a **encontrarnos** en un **token inicial** S, y la **cadena de entrada estar completamente vacía**, ya que se ha procesado entera, se acepta este resultado como árbol final detectado.

Este proceso forma la base de los algoritmos categorizados en la familia LR, y es similar para todos ellos. Sin embargo, la implementación concreta de estos algoritmos puede hacer variar en gran medida sus resultados, ya que se utilizan distintos métodos para calcular el cambio de estado (*Shift* y *Reduce*) o para realizar optimizaciones de la gramática que eviten símbolos innecesarios, símbolos no accesibles o variaciones en las reglas de producción, de manera que se obtenga el mismo resultado con menos ambigüedad.

Una de las implementaciones más habituales en los algoritmos utiliza tablas para calcular ese cambio de estados (*Shift* y *Reduce*) en base al estado anterior y la entrada siguiente. La manera en que se generan estas tablas son la principal diferencia entre los algoritmos SLR, $LALR$, CLR y GLR, que explicaremos a continuación.

Además, también se explicará como es la tabla mencionada para realizar los cambios de estados en LR, la cual sirve de base para el resto de los algoritmos de su familia y se puede crear utilizando los siguientes pasos:

Con el fin de entender donde empieza y acaba la lectura, primero se extiende la gramática añadiendo una regla con la forma "$S \rightarrow E\ eof$", donde S es el No-Terminal inicial, E, es el No-Terminal que donde se aplicarán el resto de las reglas de producción y eof, es la marca de final de lectura.

Para poder calcular qué entradas de la cadena servirán como señal de un cambio de estado (*Shift* y *Reduce*), necesitamos saber cuales son las entradas accesibles desde cada paso. Con este fin, utilizaremos un punto "$\cdot$" para marcar la posición actual en las reglas de producción.

Por ejemplo, en la regla "$A \rightarrow \alpha \cdot \beta B$" se muestra como ya se ha recibido la entrada α y se espera que la siguiente sea β para poder avanzar.

Partiendo del token No-Terminal inicial S y su regla "$S \rightarrow E\ eof$" (es decir, partiendo de $S \rightarrow \cdot E\ eof$) se calcula el **cierre** (*close* o *closure*, en inglés): $closure(S \rightarrow \cdot E\ eof)$. El **cierre** incluye la **expresión mínima de todas las reglas** (*items*) que son alcanzables para otro conjunto de reglas. De manera que, si partimos del conjunto formado únicamente por "$S \rightarrow \cdot E\ eof$", obtendríamos todas las reglas que tienen como parte izquierda E y, recursivamente, aquellas que tengan como parte izquierda el primer elemento del cuerpo (parte derecha) de E si este es otro No-Terminal.

Veamos como se construye la tabla más en profundidad utilizando como ejemplo la gramática:

- $S \rightarrow E\ eof$
- $E \rightarrow E * B$

- $E \rightarrow E + B$
- $E \rightarrow B$
- $B \rightarrow 0$
- $B \rightarrow 1$

Partiendo de la regla inicial "$S \rightarrow \cdot E\ eof$" calculamos su cierre, $closure(S \rightarrow \cdot E\ eof)$:

- $S \rightarrow \cdot\ eof$
- $E \rightarrow \cdot E * B$
- $E \rightarrow \cdot E + B$
- $E \rightarrow \cdot B$
- $B \rightarrow \cdot 0$
- $B \rightarrow \cdot 1$

La segunda, tercera y cuarta reglas se ven incluidas por la posición del punto en la primera, anterior a E. La quinta y sexta reglas se ven incluidas dado que también se ha incluido la cuarta e incluye el símbolo No-Terminal B como primer elemento del cuerpo. Este conjunto lo llamaremos estado cero S_0. Partiendo de este estado inicial debemos calcular el avance a otros estados, para ello necesitamos saber cuales son las posibles entradas desde él. Podría realizarse el mismo proceso con el conjunto de símbolos terminales y no-terminales por completo pero los cálculos ralentizarían el algoritmo en gran manera de manera innecesaria. Los símbolos de interés para el cambio de estado son aquellos en la posición siguiente al punto del estado actual, en el caso de ejemplo serían: $E, B, 0, 1$.

Partiendo del estado actual S_0 calcularemos para cada uno de los símbolos de interés anteriores el avance de las reglas con ellos.

Utilizando el símbolo 0 solo puede avanzar la regla "$B \rightarrow \cdot 0$" por lo que esta forma el estado 1, S_1:

- $B \rightarrow 0\cdot$

De igual manera, utilizando el símbolo 1 solo puede avanzar la regla "$B \rightarrow \cdot 1$" por lo que esta forma el estado 2, S_2:

- $B \rightarrow 1\cdot$

Avanzando con el símbolo B desde el estado cero S_0, se consigue el conjunto que conforma el estado 3, S_3:

- $E \rightarrow B\cdot$

Avanzando con el símbolo E desde el estado cero S_0, se consigue el estado 4,S_4:

- $S \rightarrow E \cdot eof$
- $E \rightarrow E \cdot *B$
- $E \rightarrow E \cdot +B$

Los estados S_1, S_2 y S_3 no tienen como avanzar, dado que el punto se encuentra en su última posición, pero el proceso se sigue aplicando al S_4. Partiendo del estado S_4, tenemos como símbolos de interés $*$ y $+$.

Avanzando desde S_4, con el símbolo $*$ obtenemos el conjunto S_5, que añade las reglas con la parte izquierda igual a el siguiente Token No-Terminal B:

- $E \rightarrow E * \cdot B$
- $B \rightarrow \cdot 0$
- $B \rightarrow \cdot 1$

Avanzando desde S_4 con el símbolo $+$ obtenemos el conjunto S_6:

- $E \rightarrow E + \cdot B$
- $B \rightarrow \cdot 0$
- $B \rightarrow \cdot 1$

Partiendo del estado S_5 se puede extender con los símbolos de interés $B, 0, 1$ pero para los Tokens 0 y 1 los resultados son iguales a los estados S_1 y S_2 respectivamente, para el Token B se da un nuevo estado S_7:

- $E \rightarrow E * B\cdot$

Partiendo del estado S_6 los símbolos de interés son $B, 0, 1$ y al igual que en el estado S_5 si avanzamos usando los tokens 0 o 1 iremos a los estados S_1 y S_2, para el token B se da otro nuevo estado $S_{\mathbf{8}}$:

- $E \rightarrow E + B\cdot$

Con estos **estados** y las **relaciones** entre ellos que hemos calculados se obtiene una **tabla inicial**. Como podemos ver en la tabla 2.1, en el **eje horizontal** se encuentran **todos los Tokens de la gramática** $*, +, 0, 1, E, B$ y en el **eje vertical** los **índices de los estados**. Después por cada fila buscaremos en el estado con el índice correspondiente la entrada que nos lleva a otro estado y escribiremos el índice de ese estado en la casilla correspondiente. Por ejemplo en el estado S_0 (la fila cero) al recibir un token 0 (columna 0) nos envía al estado S_1 (marcamos el índice 1).

Estados / Tokens	*	+	0	1	E	B
0			1	2	4	3
1						
2						
3						
4	5	6				
5			1	2		7
6			1	2		8
7						
8						

Tabla 2.1: Tabla inicial intermedia de cambios de estado.

La tabla final que se utiliza para los cambios de estados consta de dos partes: **acción** y ***goto***. La parte de **acción** consta de los **elementos no terminales** y **señala cuando se debe utilizar un avance en la cadena con cambio de estado** o **una reducción** utilizando las reglas de producción. La sección de *goto* se utiliza para **marcar cuál es el siguiente estado tras una reducción**.

La tabla final se construye a partir de la anterior siguiendo estos pasos:

1. **Copiar** las columnas de **No-Terminales** a la sección *goto*.
2. Copiar las columnas de **terminales** a la sección **acción** como avances.
3. Añadir una **columna** extra para el **símbolo** $, cuyo significado marca el final de la entrada a la sección de acción por cada ítem con la forma "$S \rightarrow w\ eof$".
4. Si los ítems de algún estado incluyen una regla con el punto en la posición final y el estado no es S_0, se llena la fila de este estado en la sección acción con la acción de reducción utilizando esa regla sobre el estado.

La tabla final una vez aplicados los pasos se verá como la siguiente tabla 2.2.

Estados	Acción					Goto	
	*	+	0	1	$	E	B
0			s1	s2		4	3
1	r4	r4	r4	r4	r4		
2	r5	r5	r5	r5	r5		
3	r3	r3	r3	r3	r3		
4	s5	s6			acc		
5			s1	s2			7
6			s1	s2			8
7	r1	r1	r1	r1	r1		
8	r2	r2	r2	r2	r2		

Tabla 2.2: Tabla final de máquina LR.

Los algoritmos LR funcionan empleando: la tabla final, las cadenas de entrada y una pila con los estados. Con todo ello realiza los siguientes pasos, partiendo desde el estado inicial S_0:

- Se toma el **primer elemento de la cadena**, se busca para el **estado actual (fila)** y la **entrada correspondiente (columna)** que acción se requiere.
  - Si es un **avance** s_x **se consume esa entrada** y **se añade el estado** S_{x} a la **cabeza** de la **pila de estados**.
  - Si es una **reducción** r_x no se **consume el token y se utiliza la regla** x, después **se retira el estado actual de la pila** y con el estado en la cabeza de la pila y la cabeza de la regla **se busca en la tabla el cambio de nuevo estado**.
- Esto se repite hasta llegar a un estado de aceptación.

Este **algoritmo** es **muy útil** para la **detección de algunas gramáticas** y **sirve de base para otros algoritmos** pero tiene ciertas **limitaciones**. Siendo la mayor de ellas debido a como las acciones de reducción ocupan toda la fila, por lo que se realizarán independientemente de la próxima entrada. Este caso presenta un problema cuando se crea una colisión entre una acción de cambio de estado y una reducción, o entre dos reducciones. Además, otro problema muy habitual es el conocido como "*Dangling else*", que hace referencia a una estructura básica de la programación, la sentencia *if-then-else* y como en varios lenguajes la parte *else* no es

obligatoria. Al anidar dos sentencias de este tipo, por ejemplo "*if a then if b then c else d*" se crea una ambigüedad como podemos ver en la figura 2.6 por las dos posibles interpretaciones:

- *if a then (if b then c) else d*
- *if a then (if b then c else d)*

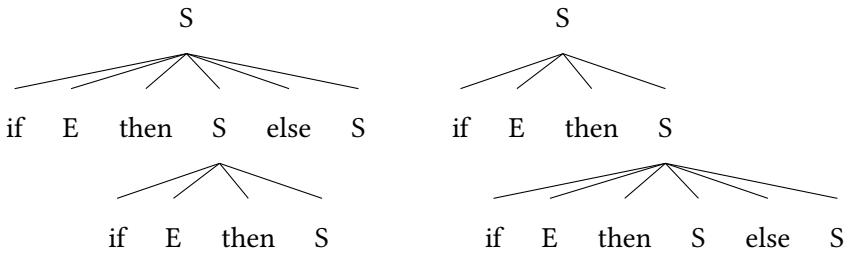


Figura 2.6: Ejemplo del problema "if-then-else" que puede formar árboles ambiguos.

SLR

El algoritmo SLR[10] (Simple LR) trata de corregir el error por colisión utilizando un post-procesado en la creación de la tabla final de transformación. Este procesado crea una tabla auxiliar llamada Siguiente (*following*, en inglés) en la que por cada símbolo No-Terminal se relaciona con un conjunto de símbolos terminales que pueden aparecer en la siguiente posición.

Por ejemplo para la gramática:

- $S \rightarrow AB$
- $A \rightarrow \alpha\beta|B$
- $B \rightarrow \alpha\beta|\gamma$

El siguiente de S es {$}, el siguiente de A es $\{\alpha, \gamma\}$ y el siguiente de B es {$}. Con estos conjuntos se crea la tabla auxiliar de siguiente:

Símbolo	S	A	B
Siguiente	$	α, $	$

Tabla 2.3: Tabla de siguientes en máquina SLR.

Para eliminar los conflictos se recorren todas las casillas con uno de la tabla de acciones. Por cada una se comprueba el conjunto siguientes de la cabeza de esa regla y, si el símbolo

terminal de la columna correspondiente a la casilla no se encuentra entre los símbolos de la tabla siguiente, se elimina la regla de reducción de la casilla. En caso contrario, se elimina la acción de avance.

Este proceso solo calcula la tabla de siguientes en base a el siguiente terminal y a la posición del No-Terminal, sin tener en cuenta los posibles estados y transiciones. Si un No-Terminal α se utiliza en varios lugares, SLR los tratará a todos de la misma manera independientemente del contexto.

CLR

El algoritmo CLR (*Canonical LR*) crea su tabla final de manera muy similar pero utiliza un **método diferente para manejar los elementos siguiente y tener en cuenta su estado**. El CLR no solo tiene en cuenta cual puede ser el próximo token Terminal si no también el estado en el que está y por los que ha pasado. Esta mejora, por otro lado, crea un gran número de estados distintos muy similares (o incluso iguales).

Por ejemplo, utilizando la siguiente gramática:

- $S \to E$
- $E \to T$
- $E \to (E)$
- $T \to n$
- $T \to +T$
- $T \to T + n$

Siguiendo el mismo proceso que el SLR y LR, el estado se calcula tomando el primer estado con la posición inicial como inicio, "$S \to \cdot E$", pero esta vez lo acompaña como segundo argumento el símbolo final \$.

Al igual que anteriormente se deben añadir al S_0 las reglas de los No-Terminales localizados justo después del marcador $\cdot$.

- $S \to \cdot E, \$$
- $E \to \cdot T$
- $E \to \cdot (E)$
- $T \to \cdot n$

- $T \rightarrow \cdot + T$
- $T \rightarrow \cdot T + n$

La primera regla tiene el segundo argumento $ pero es necesario calcularlo en las demás reglas. Este segundo argumento representa los Tokens Terminales que pueden aparecer después de la cabeza (una vez que se reduzca).

En el caso de E los posibles siguientes son:

- (por la regla $E \rightarrow (E)$
- $ por la regla $S \rightarrow E$

Para S:

- $ por la regla $S \rightarrow E$

Para T:

- $+$ por la regla $T \rightarrow T + n$
- (por la regla $E \rightarrow Tjuntocon E \rightarrow (E)$
- $ por la regla $E \rightarrow Tjuntocon S \rightarrow E$

Finalmente para obtener el set de reglas que usaremos del estado S_0 anotaremos por cada regla los elementos del campo siguiente del No-Terminal que sirve de parte izquierda, como segundo argumento. Creando varias reglas iguales con distinto segundo argumento si es preciso.

Regla 1:

- $S \rightarrow \cdot E$, $

Regla 2:

- $E \rightarrow \cdot T$, $
- $E \rightarrow \cdot T$,)

Regla 3:

- $E \rightarrow \cdot (E)$, $
- $E \rightarrow \cdot (E)$,)

Regla 4:

- $T \to \cdot n$, \$
- $T \to \cdot n$, $+$
- $T \to \cdot n$,)

Regla 5:

- $T \to \cdot + T$, \$
- $T \to \cdot + T$, $+$
- $T \to \cdot + T$,)

Regla 6:

- $T \to \cdot T + n$, \$
- $T \to \cdot T + n$, $+$
- $T \to \cdot T + n$,)

Como se puede apreciar en las reglas del ejemplo, la cantidad aumenta rápidamente, se puede simplificar su escritura uniendo las reglas y separando sus argumentos por una |

S_0

- $S \to \cdot E$, \$
- $E \to \cdot T$, \$ |)
- $E \to \cdot (E)$, \$ |)
- $T \to \cdot n$, \$ | $+$ |)
- $T \to \cdot + T$, \$ | $+$ |)
- $T \to \cdot T + n$, \$ | $+$ |)

Para calcular el estado S_1 se sigue el mismo método que en LR y SLR pero manteniendo el segundo argumento de la regla principal que invoca a las otras.

Por ejemplo, tomando como posible entrada $+$ la única regla principal que permanece es:

- $T \to + \cdot T$, \$ | $+$ |)

Pero debido a que el siguiente Token es un No-Terminal, debemos añadir todas las reglas con ese no terminal como cabeza con el símbolo de posición al principio. Y añadiremos el segundo argumento de la regla por la que las hemos añadido como segundo argumento suyo:

- $T \rightarrow + \cdot T$, $\$ \mid + \mid)$
- $T \rightarrow \cdot n$, $\$ \mid + \mid)$
- $T \rightarrow \cdot + T$, $\$ \mid + \mid)$
- $T \rightarrow \cdot T + n$, $\$ \mid + \mid)$

Este paso se repite con todos los estados hasta completarlos igual que en los algoritmos anteriores. Una vez obtenidos los estados la tabla se genera de manera similar pero en el momento de comprobar los siguientes, donde en el SLR utilizábamos la tabla auxiliar de siguientes, en este caso utilizaremos los símbolos que acompañan a la regla a reducir.

LALR

El algoritmo $LALR$[11] utiliza el mismo método que el algoritmo CLR, con una **optimización en los estados generados**. En el CLR el número de estados generados es muy elevado, para una regla por ejemplo "$S \rightarrow BC\alpha$" pueden existir múltiples versiones con distintos elementos como segundo argumento. Incluso pueden existir estados con las mismas reglas y mismos segundos argumentos, alcanzados desde distintos caminos.

Para optimizarlo se comparan estados con las mismas reglas y segundos argumentos y se unen en un estado común. Esto reduce ligeramente el número de gramáticas que el algoritmo es capaz de detectar correctamente pero optimiza en gran medida su espacio.

GLR

El algoritmo GLR[12] (de las siglas de *Generalized LR* en inglés), se basa en el funcionamiento del algoritmo LR pero **permitiendo múltiples conflictos a la vez**. Esto es posible realizando una copia de la pila hasta ese momento y añadiendo el elemento duplicado, ambas rutas se mantienen hasta que en alguna no se pueden realizar más transiciones. Además las distintas rutas dadas por una colisión pueden crear otras nuevas resultando en un número exponencial de copias. Para mejorar la eficiencia de este algoritmo se utiliza en conjunto con un $LALR$ y estructuras de grafos que permiten crear copias con una sola instancia de los elementos comunes.

2.3 Herramientas de generación

Debido a la complejidad de los compiladores y al firme proceso con el cual desarrollar sus reglas y tablas surgieron los compiladores de compiladores (o generadores de compiladores). Herramientas capaces de crear la estructura de un compilador aplicando una serie de pasos genéricos y comunes a todos. Estos generadores facilitan el análisis de un lenguaje, utilizando las reglas de producción y unas expresiones regulares como esquema para la detección de Tokens terminales.

Con el paso de los años los generadores de compiladores han avanzado y se han adaptado a las nuevas técnicas desarrolladas, pero algunos de los más populares hoy en día fueron creados hace décadas. Esta popularidad se debe en parte a la documentación ya existente sobre ellos pero también a la capacidad de los generadores antiguos para detectar gran variedad de lenguajes.

2.3.1 Lex & Flex

Lex[2] es un **generador de analizadores léxicos**, los cuales **transforman** la **entrada de texto** en el **token** adecuado **si coincide con un patrón**. Estos patrones se describen utilizando expresiones regulares. Debido a su popularidad y utilidad en el desarrollo de compiladores fue adaptado dentro del estándar *POSIX*. Lex fue desarrollado bajo una licencia privada al ser desarrollado por *Mike Lesk* y *Eric Schmidt*, más tarde se desarrollaron otras versiones de código abierto (*Open-Source*) en base al código original.

Flex[3] (*Fast Lexical Analyzer*) fue desarrollado bajo la misma idea que Lex, con una estructura casi igual en sus archivos de definición de lexemas, pero su desarrollo no proviene del código original de Lex. Flex, además, se mantiene bajo una licencia BSD.

Ambos analizadores léxicos son muy similares y compatibles con Yacc[13] y Bison[3], analizadores sintácticos de los que hablaremos a continuación. La mayor diferencia se encuentra en sus licencias, en el soporte de Flex frente a Lex y en la posibilidad de variar la cadena de entrada en el caso de Lex.

2.3.2 Yacc & Bison

Yacc[14] (de sus siglas en inglés *Yet Another Compiler Compiler*), es un **analizador sintáctico** que analiza una cadena de tokens de entrada según una serie de reglas gramáticas. Idóneamente, estos tokens pueden provenir de Lex, lo que convierte a ambos en una de las mejores herramientas para el desarrollo de lenguajes. Inicialmente fue escrito en el lenguaje de programación B pero fue reescrito varias veces hasta incluirse su implementación en C en el estándar POSIX.

El software Yacc fue creado bajo una licencia privada, al igual que Lex, y de la misma manera aparecieron implementaciones alternativas de código abierto inspiradas por él. Las más

populares fueron Byacc (*Berkeley Yacc*) de dominio público y *GNU Bison*[15] bajo la licencia GPL. Aunque inicialmente no eran compatibles con Yacc, al popularizarse se añadió en Bison dicha compatibilidad, y Byacc fue reescrito de cero para poder ser compatible.

Tanto Yacc como Bison crean máquinas LALR (explicadas en la Sección 2.2.4) capaces de detectar el código de la mayoría de lenguajes.

2.3.3 ANTLR

ANTLR[16] es un **generador de** *parsers* **actual**, que se encuentra en su cuarta versión y que sigue en desarrollo. Esta **librería** pretende ser una **herramienta única** que trabaje **sin la necesidad de Flex o Bison**. Además, genera una estructura de árbol sobre el código analizado y un elemento, al que llamaremos visitador, que recorre todo el árbol visitando cada nodo.

El código se puede añadir en una clase heredada de este visitador aplicando lo necesario cada vez que se visita el nodo adecuado. ANTLR **no utiliza ninguno de los algoritmos conocidos hasta ahora** sino una **versión propia del LL** llamado *Adaptative LL(*)* o *ALL(*)*[17].

En los últimos años ha recibido varias actualizaciones debido al número de cambios e inestabilidades generadas hasta el momento, culminando en su última versión: la versión 4, que consistió en una reescritura del algoritmo.

2.4 Compiladores

Los compiladores son **herramientas que utilizamos principalmente para la transformación de código de un lenguaje a otro** utilizando gramáticas. Como adelantábamos en la sección 2, los compiladores se pueden dividir en dos tipos principales[]: **Compiladores-Traductores** e **Intérpretes**. Ambos toman un código de entrada y analizan su contenido para obtener un resultado: en el caso de un Compilador-Traductor, ese resultado consiste en un código de salida y, en el caso de un Intérprete, en una acción. Estas definiciones son muy amplias y permitirían abarcar, por ejemplo, al *shell* de *bash* como un compilador intérprete, recibiendo código de entrada como *scripts* y analizándolo para realizar acciones en la consola. De manera informal, por otro lado, se entiende como compilador aquellos programas que toman código como entrada y lo analizan para obtener sus correspondientes instrucciones en lenguaje máquina o en lenguaje intermedio (Byte-Code). De manera informal, se entiende como intérpretes a aquellos que toman el código como entrada para aplicar las operaciones correspondientes dentro de un entorno virtual. Sin embargo, estas simplificaciones dejan de lado el caso de los compiladores que transforman el código en otro en un lenguaje distinto como por ejemplo: *Pandoc*, un conversor de documentos que puede recibir un documento en lenguaje markdown y convertirlo en HTML, LaTeX, docx etc...

Todos los compiladores suelen tener en común **una estructura para representar el**

resultado del análisis sintáctico y **utilizarla para poder recorrer y optimizar el código al transformarlo**. Esta estructura es, por su similitud con la representación del código, es **un árbol**, compuesto de nodos.

2.4.1 Árbol Sintáctico

Durante el proceso de análisis **se recorre el árbol generado por el** *parser*, el cual se conoce como **árbol sintáctico**[18] (o en inglés *Abstract syntax tree* (AST)). Podemos ver un ejemplo en la figura 2.7. Este árbol contiene la información del código necesaria para transformarlo, una vez completada dicha transformación, se asegura de que la estructura es adecuada y correcta con la gramática del lenguaje, siendo comprobado gracias al parser. Desde la cabeza del árbol, donde se encuentra la máxima representación del documento, se baja recorriendo cada rama, en ellas podemos encontrar entre otros datos: el texto original, el token correspondiente y una lista de nodos descendientes en el caso de ser un token No-Terminal.

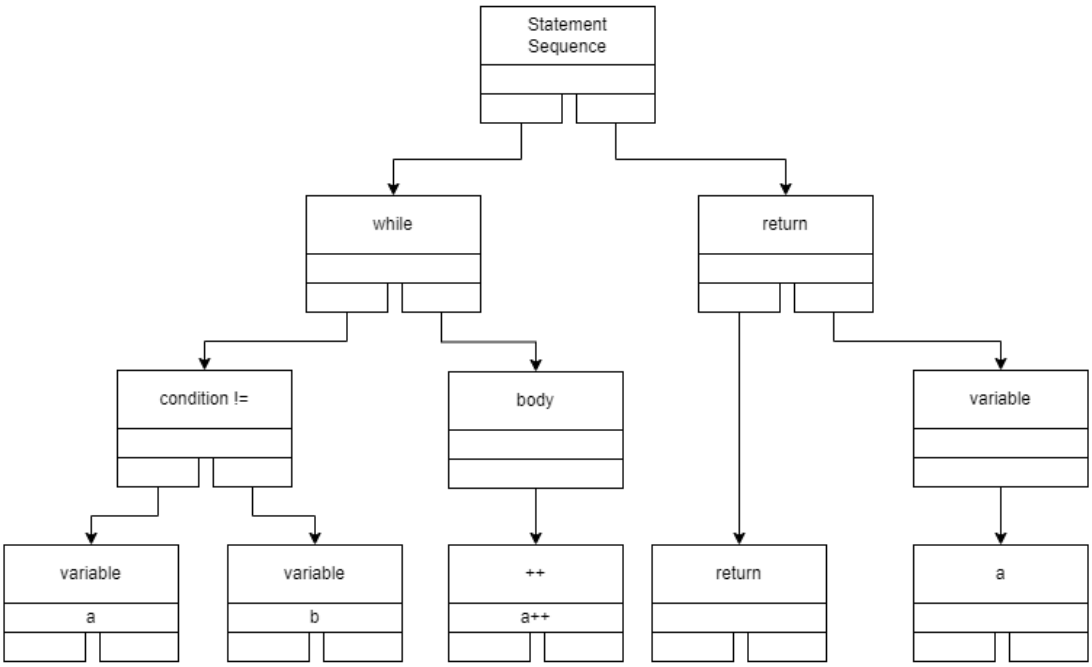


Figura 2.7: Ejemplo de un árbol abstracto.

2.4.2 Compilación

Los pasos principales de un compilador se pueden dividir en tres secciones. La **primera sección** es la llamada en inglés *front-end*, en ella se convierte la entrada de código a un árbol sintáctico abstracto a través de los procesos anteriormente explicados, y se recorre dicho árbol para comprobar los tipos en aquellos lenguajes con tipado estático o en busca de errores.

Una vez completado este segmento se pasa a la **segunda sección**, nombrado en inglés *middle-end* o **parte intermedia**. En este segmento se realizan optimizaciones independientes de la CPU, entre ellas: eliminación de código no alcanzable o inútil, optimización de constantes, relocalización de segmentos fuera de bucles si es posible, etc.

Tras esto, el árbol resultante se envía al **último segmento o sección** llamado en inglés *back-end*. En él se pueden realizar otras optimizaciones, esta vez dependientes de la CPU para el código. La finalidad de el *back-end* es construir una serie de instrucciones comprensibles en lenguaje máquina que representen el código para poder ser ejecutadas. Estas instrucciones solo podrán ser ejecutadas en sistemas con una configuración y capacidades iguales o compatibles, por ende su distribución es limitada.

Las optimizaciones realizadas dependen en parte del lenguaje utilizado y de su funcionamiento, algunas son muy habituales como las realizadas por el *middle-end* pero podrían desarrollarse optimizaciones exclusivas de un lenguaje debido a características especiales propias de él. Como por ejemplo, la recursividad terminal (conocida en inglés como *Tail-recursion*) propia de lenguajes funcionales como OCaml, Elixir, etc.

2.4.3 Interpretación

Un **intérprete** funciona de manera muy **similar a un compilador-traductor**, siendo **su principal diferencia el** *back-end*. En su caso **no realiza la transformación** a instrucciones máquina propias de la CPU o código intermedio, si no a **acciones ejecutables en el entorno virtual**.

Para poder ejecutar las acciones y obtener los resultados acordes, es necesario disponer en la sección de *back-end* un **procesamiento** que, en base a determinadas entradas, como resultado produzca unos cambios. Este elemento sería el encargado de analizar la entrada y manejar los recursos a su disposición para realizar la acción, ya sean estos memoria, directorios, archivos u otros elementos. Una interfaz de línea de comandos, una vez recibe un comando en concreto, por ejemplo *Bash* al recibir una orden "ls", debe acceder a la lista de directorios para la posición actual y mostrarla a través del terminal. Pero ese mismo intérprete al recibir el comando "$echo$" con un archivo debe acceder al interior del documento y cambiarlo.

Los intérpretes pueden recibir un archivo de texto correspondiente al programa a analizar y ejecutar o pueden utilizar un bucle interactivo. Los bucles interactivos reciben una línea en su entrada de cada vez, la analizan y ejecutan para finalmente esperar por la siguiente. Muchos lenguajes interpretados como Python, Elixir u OCaml utilizan este bucle además de tener la opción de enviar el código completo al intérprete.

2.4.4 Traducción

Los **compiladores-traductores** pueden **traducir** a cualquier **lenguaje de salida**, como por ejemplo los compiladores que traducen a código máquina. Este tipo de compiladores-traductores pueden requerir no solo optimizaciones si no el desarrollo de una serie de equivalencias entre los dos lenguajes que serán utilizadas para poder traducir el código.

Código Máquina

El **código máquina**[19], conocido en inglés como *Machine code*, son las **instrucciones procesables por un modelo de CPU** concreto para realizar las acciones más básicas disponibles, como: cargar memoria, escribir en registros o leer de ellos. Este código es el nivel más bajo y básico de operaciones que se pueden realizar, su contenido es estrictamente de números representados de forma binaria. Al conjunto de instrucciones comprensibles por una *CPU* se les llama *set* de instrucciones. Existen distintos tipos de *sets* de instrucciones, los tipos *RISC* (por sus siglas en inglés *Reduced Instruction Set Computer*), como por ejemplo *MIPS* (*Microprocessor without Interlocked Pipelined Stages*) o *ARM* (*Advanced RISC Machines*), que se caracterizan por instrucciones muy simples y de bajo consumo ideales para dispositivos diseñados para aprovechar la batería. Por otro lado, existen los *set* tipo *CISC* (*Complex Instruction Set Computer*), como el popular $x86$ (y sus versiones de 32 y 64 bits), que se caracterizan por contener instrucciones basadas en operaciones complejas.

Código intermedio (Byte-Code)

Las instrucciones contenidas en el código intermedio (*Byte-Code*), popularizadas por su uso en la máquina virtual de Java, son **instrucciones ficticias** que **emulan el código máquina**. Sin embargo, **no se ejecutan directamente en un procesador real** si no que estas instrucciones son analizadas y procesadas por la máquina virtual, permitiendo la distribución de programas para esa misma máquina de manera sencilla.

2.5 Máquinas Virtuales

Las **máquinas virtuales**[20] en el contexto de lenguajes de programación se refiere a **entornos virtuales en los que se ejecutan determinadas acciones** correspondientes con órdenes o instrucciones. Estos entornos se comportan como programas que **manejan y controlan**, entre otros: la **memoria**, los **hilos de ejecución** y los **recursos dedicados a la ejecución** de uno o varios procesos.

En este documento nos referiremos a este tipo de entornos como máquinas virtuales o *VM* (siglas de su expresión en inglés *Virtual Machine*) salvo que se especifique de otra manera.

También existen máquinas virtuales capaces de **comportarse como una copia virtual exacta de una máquina real** con todos sus periféricos y componentes. Estas máquinas utilizan el mismo sistema operativo que una máquina real y son a efectos prácticos tan capaces como ellas. Debido a la virtualización **su eficiencia es mucho menor** pero otorga la posibilidad de **contener todo un entorno** dentro del propio ordenador y copiarlo o distribuirlo a gran escala. Los manejadores de máquinas virtuales permiten exportar o pausar la máquina para su análisis entre otras ventajas.

Las máquinas virtuales como entornos de ejecución tienen un propósito claro, la **portabilidad** y **facilidad de compilación**. Debido a que el código de el lenguaje de programación se ejecuta en un entorno virtual dentro de la máquina, la **incompatibilidad con instrucciones** o especificaciones de *hardware* no existen. La única **compatibilidad necesaria** es entre el propio entorno virtual y la máquina. La complicación que asume una compilación solo se debe llevar a cabo una vez y para un solo proyecto, el entorno virtual, y esto lo afrontan los desarrolladores/as del propio entorno virtual **no los programadores** que utilicen el lenguaje. De esta manera al programar en estos lenguajes se asume una **perdida de eficiencia** por el entorno virtual pero una gran ventaja en **portabilidad** del proyecto. Ejemplo de ellos es desarrollar para móvil o para web en JAVA, es la máquina virtual quien asume la compatibilidad con dispositivos, existiendo diferentes versiones de la misma para cada uno.

Las máquinas virtuales se pueden dividir en dos tipos: aquellas que **compilan el lenguaje a un árbol abstracto** y ejecutan las acciones a ese árbol y, aquellas máquinas que **reciben un a serie de pseudo-instrucciones** de un código ya compilado para realizar las acciones de manera más rápida. Este último caso es, por ejemplo, el de lenguajes como JAVA que realizan una compilación a *Byte-Code*, una serie de pseudo-instrucciones para su máquina virtual JVM (de sus siglas en inglés *Java Virtual Machine*), que después sirven de entrada para la ejecución. De esta manera cualquier código una vez compilado puede ser distribuido a una máquina real capaz de ejecutar el entorno virtual.

Capítulo 3

Metodología

Una de las etapas más críticas de un proyecto es su planificación. Esto se debe a que la misión de esta etapa, entre otras, es ayudar a comprender cuáles son las necesidades del mismo y cuáles son las tareas que lo conforman. Por ello, este capítulo se dividirá en las siguientes secciones: **metodología seguida**, **desglose de las tareas**, **recursos necesarios** y **estimación de costes**.

3.1 Metodología

La metodología escogida se ha basado en dos características principales: **incrementalidad**, ya que nos permitirá ir desarrollando funcionalidades que irán aumentando desde más básicas hasta llegar al producto completo, e **iteratividad**, que nos facilitará el disponer de productos funcionales parciales. Esto nos concede mayor flexibilidad y mejor gestión de errores en comparación a otras metodologías más rígidas, como la metodología en cascada, donde hasta el final no se obtiene un producto funcional.

Cada iteración contará con un análisis, un diseño, una implementación y, finalmente, las pruebas, como podemos observar en la figura 3.1.

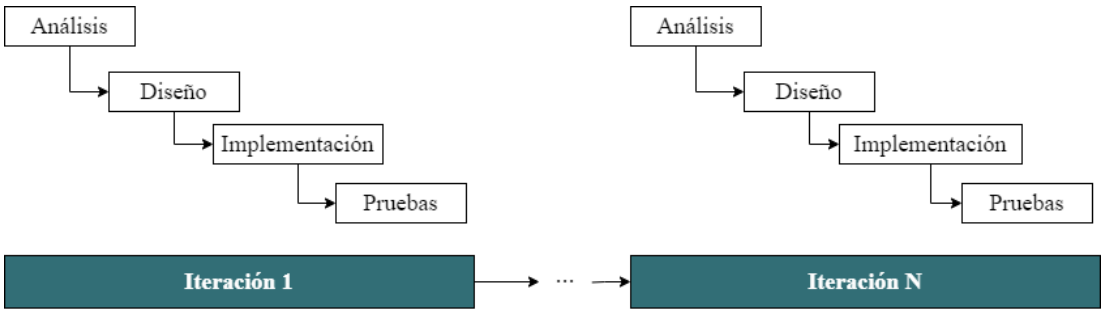


Figura 3.1: Ejemplo de la metodología escogida.

3.2 Desglose de las tareas

Para la planificación y el desglose de las tareas se ha creado el **diagrama de Gantt** que puede verse en la Figura 3.2. Gracias a este tipo de cronograma podremos visualizar, en conjunto y a alto nivel, la estimación temporal y las relaciones entre las tareas. Cabe destacar que, el diagrama tendrá la estructura mencionada en la figura 3.1, donde las tareas dentro de cada iteración pueden pertenecer a las cuatro categorías previamente mencionadas.

3.3 Recursos

Para facilitar la reproducibilidad de este proyecto, en esta sección se procederá a listar cuales han sido los recursos necesarios para llevarlo a cabo que, según su naturaleza, se dividirán en dos tipos: **recursos humanos** y **recursos materiales**.

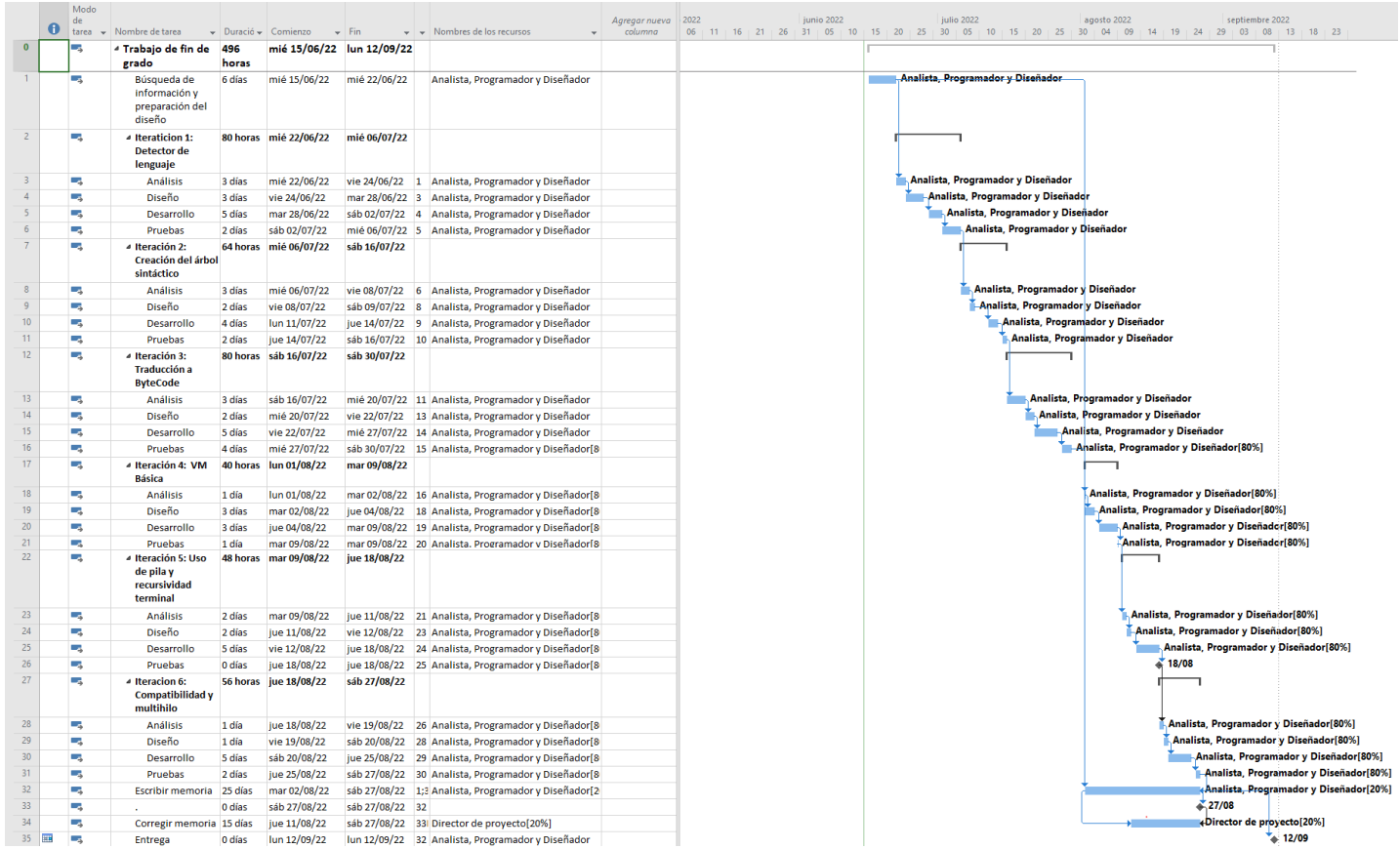
3.3.1 Recursos humanos

Respecto a los **recursos humanos**, este proyecto constará principalmente de cuatro roles: **jefe de proyecto, analista, diseñador y programador**. El papel de **jefa de proyecto** corresponderá a la **directora de proyecto**, la cual se ha encargado de la supervisión y la guía del trabajo. El **resto de papeles** debido al carácter individual del proyecto, corresponderán al alumno.

3.3.2 Recursos materiales

Respecto a los **recursos materiales**, en este apartado se han dividido en función de si pertenecen a la categoría del *hardware* o del *software*:

- **Recursos *hardware***
  - Ordenador con un sistema operativo MacOs en versión 12.5.1 o superior y 4GB de memoria RAM.
- **Recursos *software***
  - En cuanto al compilador
    - Python 3.9
  - En cuanto a la máquina virtual
    - make 3.81
    - g++ 12.1.0



	Modo de tarea	Nombre de tarea	Duració	Comienzo	Fin		Nombres de los recursos
0		**Trabajo de fin de grado**	**496 horas**	**mié 15/06/22**	**lun 12/09/22**		
1		Búsqueda de información y preparación del diseño	6 días	mié 15/06/22	mié 22/06/22		Analista, Programador y Diseñador
2		**Iteraticion 1: Detector de lenguaje**	**80 horas**	**mié 22/06/22**	**mié 06/07/22**		
3		Análisis	3 días	mié 22/06/22	vie 24/06/22	1	Analista, Programador y Diseñador
4		Diseño	3 días	vie 24/06/22	mar 28/06/22	3	Analista, Programador y Diseñador
5		Desarrollo	5 días	mar 28/06/22	sáb 02/07/22	4	Analista, Programador y Diseñador
6		Pruebas	2 días	sáb 02/07/22	mié 06/07/22	5	Analista, Programador y Diseñador
7		**Iteración 2: Creación del árbol sintáctico**	**64 horas**	**mié 06/07/22**	**sáb 16/07/22**		
8		Análisis	3 días	mié 06/07/22	vie 08/07/22	6	Analista, Programador y Diseñador
9		Diseño	2 días	vie 08/07/22	sáb 09/07/22	8	Analista, Programador y Diseñador
10		Desarrollo	4 días	lun 11/07/22	jue 14/07/22	9	Analista, Programador y Diseñador
11		Pruebas	2 días	jue 14/07/22	sáb 16/07/22	10	Analista, Programador y Diseñador
12		**Iteración 3: Traducción a ByteCode**	**80 horas**	**sáb 16/07/22**	**sáb 30/07/22**		
13		Análisis	3 días	sáb 16/07/22	mié 20/07/22	11	Analista, Programador y Diseñador
14		Diseño	2 días	mié 20/07/22	vie 22/07/22	13	Analista, Programador y Diseñador
15		Desarrollo	5 días	vie 22/07/22	mié 27/07/22	14	Analista, Programador y Diseñador
16		Pruebas	4 días	mié 27/07/22	sáb 30/07/22	15	Analista, Programador y Diseñador[8
17		**Iteración 4: VM Básica**	**40 horas**	**lun 01/08/22**	**mar 09/08/22**		
18		Análisis	1 día	lun 01/08/22	mar 02/08/22	16	Analista, Programador y Diseñador[8
19		Diseño	3 días	mar 02/08/22	jue 04/08/22	18	Analista, Programador y Diseñador[8
20		Desarrollo	3 días	jue 04/08/22	mar 09/08/22	19	Analista, Programador y Diseñador[8
21		Pruebas	1 día	mar 09/08/22	mar 09/08/22	20	Analista, Programador y Diseñador[8
22		**Iteración 5: Uso de pila y recursividad terminal**	**48 horas**	**mar 09/08/22**	**jue 18/08/22**		
23		Análisis	2 días	mar 09/08/22	jue 11/08/22	21	Analista, Programador y Diseñador[8
24		Diseño	2 días	jue 11/08/22	vie 12/08/22	23	Analista, Programador y Diseñador[8
25		Desarrollo	5 días	vie 12/08/22	jue 18/08/22	24	Analista, Programador y Diseñador[8
26		Pruebas	0 días	jue 18/08/22	jue 18/08/22	25	Analista, Programador y Diseñador[8
27		**Iteracion 6: Compatibilidad y multihilo**	**56 horas**	**jue 18/08/22**	**sáb 27/08/22**		
28		Análisis	1 día	jue 18/08/22	vie 19/08/22	26	Analista, Programador y Diseñador[8
29		Diseño	1 día	vie 19/08/22	sáb 20/08/22	28	Analista, Programador y Diseñador[8
30		Desarrollo	5 días	sáb 20/08/22	jue 25/08/22	29	Analista, Programador y Diseñador[8
31		Pruebas	2 días	jue 25/08/22	sáb 27/08/22	30	Analista, Programador y Diseñador[8
32		Escribir memoria	25 días	mar 02/08/22	sáb 27/08/22	1;3	Analista, Programador y Diseñador[2
33		.	0 días	sáb 27/08/22	sáb 27/08/22	32	
34		Corregir memoria	15 días	jue 11/08/22	sáb 27/08/22	33	Director de proyecto[20%]
35		Entrega	0 días	lun 12/09/22	lun 12/09/22	32	Analista, Programador y Diseñador

Figura 3.2: Diagrama de Gantt.

Respecto a los recursos humanos asignados a las tareas, es importante destacar que, para el programador no se ha contado con lo que sería una jornada laboral habitual (*8h/día* excepto fines de semana) sino que se ha empleado un horario más compatible con su situación: *6h/día* por semana y *6h/día* los sábados.

3.4 Estimación de costes

En este proyecto se han tenido en cuenta principalmente dos tipos principales de costes: los **materiales** y los **asociados a los recursos humanos**. A continuación se mostrará un desglose de los mismos, y se detallará cómo se han calculado.

En cuanto a los costes pertenecientes a la categoría de recursos humanos, se calcularán en función del rol que se esté a desempeñar. Para ello, se han tomando como referencia los datos proporcionados en la página Glassdoor. Dando como resultado la tabla 3.1.

	Salarios	horas x humano	Inversión
Director de proyecto	30€/h	18h	540€
Analista, diseñador, programador	18€/h	414h	7.452€
		Total:	7.992€

Tabla 3.1: Estimación de coste de los recursos humanos.

Respectos a los **costes de los recursos hardware**, aunque se podrían tener en consideración el ordenador, se ha dejado como nulo, ya que ha sido el programador quien lo ha aportado. Respecto a los recursos software, al ser de licencia abierta y sin precio asociado tampoco cuentan como gasto. Por lo tanto, el **coste total del proyecto** es de 7.992€.

Capítulo 4

Desarrollo del trabajo

El lenguaje que se ha implementado se llama **TedLang**: se utiliza el sufijo *Lang*, debido a el término en inglés *Language*, para marcarlo como un lenguaje de programación, y Ted, como diminutivo de *Teddy*, nombre al que se refieren en inglés habitualmente a los osos de peluche. Muchos de los lenguajes actuales tienen una **mascota asociada**: la serpiente de Python, la taza de café de JAVA, el cangrejo (no-oficial, pero muy popular) de Rust y la taltuza de Go, como se puede ver en la figura 4.1. Por lo que adoptar una mascota propia, era adecuado darle personalidad al lenguaje.

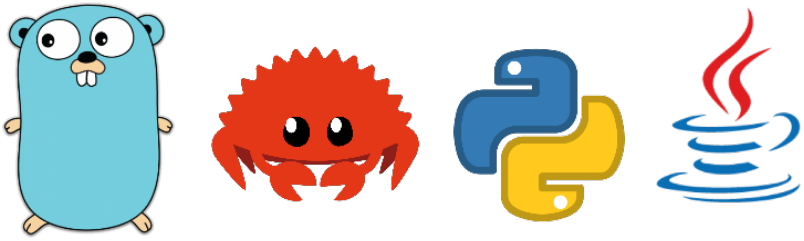


Figura 4.1: Iconos de los lenguajes, de izquierda a derecha: Go, Rust, Python, Java.

Tedlang se ha centrado en **ser un lenguaje simple y fácil de aprender**, un **lenguaje funcional de toma de contacto** para aquellos desarrolladores/as que se acercan por primera vez a este paradigma (ya sea desde cero o desde otros lenguajes más conocidos). Por ello, se ha escogido un oso de peluche como mascota, agradable y un referente de infancia. Sus recursos y diseño giran entorno a esto, y como veremos en este apartado, las decisiones respecto a su sintaxis y características principales también.

Existen una gran variedad de lenguajes en el mundo de la programación, cada uno con diferentes propiedades que sirven a un fin específico. Según propiedades como su paradigma o la sencillez y eficiencia, estos lenguajes se pueden dividir en varios grupos.

Respecto a su sencillez y eficiencia se pueden crear dos grupos de lenguajes: los lenguajes de **bajo nivel** centrados en la **eficiencia** y robustez de sus sistemas, y los lenguajes de **alto nivel** usualmente repletos de **utilidades** y herramientas disponibles.

Respecto a sus paradigmas, los modelos que definen las principales características de estos lenguajes. Existen principalmente tres paradigmas, lenguajes **orientados a objetos**, **lenguajes funcionales** y **lenguajes declarativos**.

El primer grupo, lenguajes orientados a objetos, está centrado en la mensajería entre clases (representaciones como objetos de los elementos del programa). El segundo grupo toma su idea de las funciones matemáticas, recibiendo un valor para devolver otro. En cuanto al último grupo se conforma por una serie de declaraciones continuadas para describir el programa.

Aun teniendo en cuenta estas divisiones, los lenguajes existen en un espacio en desarrollo y por lo tanto unos influencian a otros, dando pie a la creación de lenguajes **multiparadigma**, con elementos de más de un tipo.

A continuación, expondremos los lenguajes que han servido de referencia para el diseño e implementación de TedLang, el lenguaje desarrollado en este proyecto, y tras ello ahondaremos en la relación entre estos y las características del mismo.

4.1 Referentes

Estos principales referentes han servido de base y apoyo en la estructura y funcionamiento del lenguaje desarrollado, tomando elementos característicos de cada uno y viéndose fuertemente influenciado.

4.1.1 Python

Python es un lenguaje creado pensando en la sencillez y legibilidad, varios de sus principios en el "Zen de Python"[21] hacen referencia a ello directa o indirectamente. Es un lenguaje ejecutado sobre una máquina virtual lo que lo hace fácilmente portable y, aun que afecta negativamente a su eficiencia, apoya sus valores como lenguaje en cuanto a sencillez. La propia máquina virtual que ejecuta el código es capaz de reconocer los tipos correspondientes a las variables y manejar la memoria con el recolector de basura incorporado, eliminando ese peso del desarrollador y facilitando su trabajo.

No es necesario declarar los tipos utilizados por cada variable o función en Python, pero debido a la legibilidad que aportan se añadieron anotaciones de tipos[22] opcionales en la versión 3.5. Desde entonces han avanzado con cada actualización, en la Figura 4.2 se muestra un código ejemplo de una función.

La sintaxis de este lenguaje refleja muy bien su sencillez. Como se ve en la Figura 4.2 las funciones se definen con una palabra reservada "def" inicial, seguida de un esquema

```
    def greeting(name: str) -> str:
        return 'Hello ' + name
```

Figura 4.2: Función con tipado explicito en Python 3.10.

ya conocido en varios lenguajes compuesto por el nombre de la función, sus argumentos acompañados de los tipos correspondientes y el tipo de variable que se espera de vuelta. El lenguaje evita símbolos poco habituales como "{" "}" si es posible para evitar el ruido visual en el código.

4.1.2 Elixir

Elixir[23] es un lenguaje funcional, enfocado en la escalabilidad y mantenibilidad. Este lenguaje se beneficia de utilizar funciones muy cortas y eficientes que se apoyan en la recursividad terminal que implementa su sistema. Además implementa un sistema de mensajería entre diferentes procesos internos que permite enviar mensajes entre ellos, estos procesos son manejados automáticamente por el manejador de procesos interno maximizando la estabilidad. La sintaxis del lenguaje, como se ve en el ejemplo de la Figura 4.3 es muy sencilla y descriptiva, los módulos se definen con la palabra clave "$defmodule$" u las funciones con "def". El sistema también es capaz de deducir los tipos de las variables y manejar la memoria por si mismo.

```
defmodule Math do
       def sum(a, b) do
         a + b
       end
     end
```

Figura 4.3: Ejemplo de una función en Elixir.

4.1.3 Java

Java es un lenguaje de programación orientado a objetos, basado en el uso de objetos como representación de los elementos del código. Estos objetos llamados "Clases" se comunican por medio de sus métodos, funciones relacionadas intrínsecamente con la clase, de esta manera es el objeto el que ejecuta el método. El código queda organizado por módulos y clases, estas clases contienen los atributos y funciones relacionadas, dando una estructura comprensible y clara. En otros lenguajes sin esta organización las funciones de un módulo repiten los argumentos en cada con un orden concreto, cargando en el desarrollador el peso de mantener esta convención a mano. En Java siempre puedes acceder al objeto contenedor de la función actual a través de

la palabra clave "*this*". El ejemplo de la Figura 4.4 muestra una clase de ejemplo con atributos en la que se puede apreciar la estructura y en la Figura 4.5 se puede apreciar como se realiza la llamada a un método para un objeto de esa clase.

```
public class Coche {
  int ruedas = 4;
  int puertas = 5;
  int marchas = 5;
  double velocidad = 5;
  String[] matricula;

  public coche(String[] matricula) {
   this.matricula = matricula;
  }

  void frenar( ) {
   this.velocidad = 0;
  }

  void acelerar( ) {
   this.velocidad = this.velocidad + 1;
  }
}

```

Figura 4.4: Ejemplo de una clase en Java.

```
 cocheDelAbuelo.frenar();

```

Figura 4.5: Llamada a un método de la clase Coche anterior.

4.2 Diseño

El **diseño es uno de los aspectos principales y más importantes** a la hora de crear un lenguaje de programación. Aspectos como su finalidad, los detalles de la implementación o su gramática, son algunos de los elementos que pueden variar drásticamente según el uso que se le dé al lenguaje.

Al igual que con Python, TedLang se centra principalmente en la **facilidad de aprendizaje** y la **simpleza**. Sus definiciones y utilidades deben **evitar los símbolos complejos o inesperados** dentro de lo posible, **evitar efectos colaterales** o con **varios usos no explícitos**. Además, las estructuras comunes y habituales de las funciones se mantendrán dado que facilitan la legibilidad y comprensión del código en el contexto de la programación. De esta manera, por ejemplo, la estructura para una función se define como aquella mostrada en la figura 4.6, donde

"fun" marcará la declaración de una función, **evitando la verbosidad de otros lenguajes** (como se aprecia en la figura 4.7 donde se aprecia la estructura de una función de Java).

```
fun nombre_de_la_funcion ( tipo_1 nombre_1) tipo_2:
    1 + 1
endfun

```

Figura 4.6: Código de ejemplo de una función en TedLang.

```
public static void main( String args[]){
    // codigo
}
```

Figura 4.7: Firma de la función principal en Java.

Como podemos ver, en la Figura 4.6, la firma de la función seguirá el orden de elementos y convenciones de otros lenguajes. Los argumentos, al igual que en otros lenguajes y con el fin de mantener la similitud con ellos, estarán rodeados por paréntesis y separados por comas, con una indicación de su tipo anterior a su nombre. Tras el cierre de paréntesis, se señala el tipo que devolverá la función marcando el fin de la firma de la función con dos puntos. De esta manera la lectura del código será más sencilla y explicita.

A la hora de nombrar una función se admite cualquier identificador que contenga letras (mayúsculas o minúsculas), barras bajas y números; pero se recomienda el uso de snake case para mejorar su legibilidad.

TedLang es un **lenguaje funcional**, debido a esto comparte varios de sus mecanismos y principios con otros lenguajes funcionales. Este lenguaje toma gran inspiración de el lenguaje Elixir, beneficiándose de las funciones cortas. Para aprovechar al máximo su eficiencia, se ha implementado un sistema de recursividad terminal. Este lenguaje fomenta un uso mayoritario de expresiones (elementos con valor de retorno) en su código, volviendo las funciones una gran operación que asemeja a las funciones matemáticas. Como se comentó anteriormente, los lenguajes funcionales como Elixir tan solo cuentan con los módulos como estructuras capaces de organizar el código, dejando en manos de los desarrolladores/as la tarea de controlar el orden de los argumentos. Esto crea un código repetitivo y verboso, como se puede apreciar en la Figura 4.8.

Para este lenguaje hemos adaptado esta estructura de manera que no entre en conflicto con los principios funcionales del lenguaje pero sirva como una herramienta de organización y simplificación del código y sus funciones.

Al igual que Elixir y Java, **el código se puede estructurar en Módulos** para organizarlo, pero además, se **implementa el uso de clases** como estructuras para mantener funciones y

```
defmodule Coche do

  def acelerar(coche) do
    %{matricula: coche[:matricula],
        velocidad: coche[:velocidad] + 1}
  end

  def frenar(coche) do
    %{matricula: coche[:matricula], velocidad: 0}
  end

  def crear_coche(codigo_matricula) do
    %{matricula: codigo_matricula, velocidad: 0}
  end
end
```

Figura 4.8: Ejemplo en Elixir de un módulo con funciones de firma similar.

elementos comunes dentro de un mismo elemento.

4.2.1 Características especiales

Como ya hemos comentado, TedLang se centra en la simpleza, facilidad y funcionalidad, para conseguirlo hemos desarrollado varias características principales, en las que profundizaremos a continuación. Siendo estas:

- Organización por Módulos y Clases
- Entorno Virtual
- Compatibilidad con otros entornos virtuales
- Recursividad Terminal

4.2.2 Organización en Módulos y Clases

Como adelantábamos anteriormente, el código de TedLang no solo se agrupa en módulos, como otros lenguajes funcionales, también utiliza la estructura de clases y herencia entre ellas para dar una mejor forma a la relación entre funciones y objetos.

En la figura 4.9 se encuentra un ejemplo del código utilizado para un módulo que contiene una clase. Las clases pueden **contener atributos y métodos** al igual que en Java, de manera que **sirva como una estructura de datos** para la organización. Python también permite crear clases pero las estructuras no son fijas y están abiertas al cambio, añadiendo dinámicamente campos si es necesario, Tedlang sigue la dirección de Java **utilizándolas como algo invariable**

```
import Math;
import Logic;

Mod Ejemplo as

    fun nombre_funcion1( int a, int b) double:
      a + b
    endfun

    fun nombre_funcion2(str c, str d) str:
      let g = 'hola mundo' in
        if c == 'a' then
          c + d
        else
          g
    endfun

    Class Coche as

        let velocidad = 0 in
        let ruedas = 4 in

            fun acelerar():
                this.velocidad = this.velocidad + 1
            endfun
            fun frenar():
                this.velocidad = 0
            endfun

    endclass

endmod

fun carreras(Coche deportivo, Coche turismo):
    if deportivo.velocidad > 100 then
        deportivo.frena() &
        deportivo
    else
        if turismo.velocidad > 100 then
            turismo.frena() &
            turismo
        else
            deportivo.frena() &
            deportivo
endfun
```

Figura 4.9: Ejemplo en TedLang de un módulo con funciones y una clase.

y bien definido. Durante la implementación trataremos los detalles de el uso de memoria y manejo de clases en comparación con JAVA.

4.2.3 Recursividad Terminal

En el sistema implementamos una **optimización especial llamada recursividad terminal[24]** (también llamada *tail recursion* en inglés). En el transcurso de una función o método es posible que esta se **llame a sí misma** incurriendo en lo que llamamos **recursividad**. Cuando esto ocurre en un sistema sin la optimización **los datos de la función original quedan grabados en la memoria mientras se ejecuta la nueva llamada** y si se realiza demasiadas veces puede **llenar la memoria hasta desbordarla**. A esto lo llamamos stack overflow).

La recursividad terminal optimiza las llamadas de manera que, si **el último paso de la función es una llamada recursiva, se utiliza la memoria ya reservada** de su anterior ejecución. Debido a que es el último paso, **la memoria reservada no es útil** y en lugar de liberarse es **reutilizada por la nueva función**.

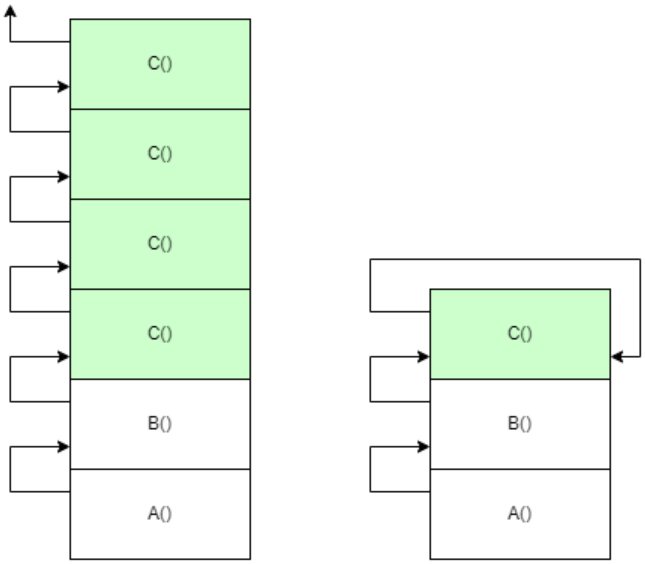


Figura 4.10: Ejemplos del uso de memoria sin recursividad terminal (izquierda) y con ella (derecha).

Esto nos permite optimizar el código creando **funciones muy cortas y eficientes** en el uso de memoria, como se aprecia en la figura 4.10.

4.2.4 Entorno Virtual

Con el fin de hacer el uso del lenguaje más **accesible y portable** para todos los nuevos usuarios, hemos diseñado un entorno virtual sobre el que ejecutar estos programas. Esta decisión tiene mucho peso en la **eficiencia** del lenguaje pero también en la **facilidad y flexibilidad** que le otorga. De manera natural, **los programas que se ejecutan a través de un entorno virtual son más lentos que sus iguales en código máquina**, ya que el entorno utiliza parte del tiempo en ejecución para analizar la entrada y realizar los procesos necesarios de cada ciclo de ejecución virtualmente[25]. Esta perdida de eficiencia **se puede suavizar con una compilación a Byte-code** que transforme el código en una serie de instrucciones lo más simple posibles.

La máquina virtual **está escrita sobre otro lenguaje**: C++, por lo que Tedlang será ejecutable en aquellas máquinas con **soporte para un compilador C++** con poca o ninguna dificultad. Este lenguaje **es uno de los más extendidos** en el desarrollo software (22.5% según la encuesta de StackOverflow de 2022[26]) y es uno de los más utilizados en sistemas empotrados o en desarrollo de sistemas, junto con C y Rust.

La máquina virtual de TedLang **nos otorga además un control con muchas posibilidades y flexibilidad** tanto para buscar errores como para **obtener información** sobre los procesos en su interior.

4.2.5 Compatibilidad con otros entornos virtuales

Con el fin de ampliar las posibilidades del lenguaje, en este trabajo se ha planteado una capa de compatibilidad con máquinas virtuales más conocidas. Esto permite aprovechar funcionalidades mejor implementadas por máquinas externas que han tenido más tiempo para su desarrollo, permitiendo mantener un sistema más robusto de manera más sencilla. Las máquinas virtuales escogidas son **JVM** (la máquina virtual de Java) y **BEAM** (la máquina virtual del lenguaje Elixir y Erlang), por sus similitudes con el lenguaje en **estructura de clases** y paradigma **funcional** respectivamente.

4.3 Análisis

Como trazo general del lenguaje, se puede apreciar que tanto las funciones como clases y módulos **acaban con una variación de la palabra "end"** que marca el segmento que finaliza, de manera que es **más fácil de distinguir los elementos una vez anidados**. Las expresiones "if" siempre van acompañados con un bloque "else", de esta manera aseguramos que **un valor siempre sea devuelto en la expresión**.

Como lenguaje funcional, la idea principal se apoya en las **funciones matemáticas**, que dado un valor de entrada devuelven uno de salida. Esto es algo que hemos intentado reforzar

en la implementación de este lenguaje, de manera que **todas las funciones tienen una única expresión como cuerpo**, y es su resultado lo que devuelven. Dicha función puede ser muy compleja y recurrir a diferentes tipos de llamadas, como en el ejemplo de la función $carreras$ en la figura 4.9, pero es ante todo una única expresión.

Durante el apartado sobre la implementación trataremos en detalle como estos principios toman parte en la interpretación y ejecución del código.

4.4 Compilador

El compilador, **encargado de la detección del código y su transformación** en las instrucciones intermedias adecuadas, es una parte fundamental del desarrollo. Como se ha descrito antes, existen varias herramientas cuyo fin es el de crear máquinas capaces de detectar gramáticas. Durante el desarrollo de este proyecto se **realizaron distintas pruebas con las herramientas más conocidas disponibles**: Flex, Bison y ANTLR. Tras varios experimentos utilizando gramáticas básicas se ha demostrado que es una opción más útil realizar una **implementación propia** del compilador.

Esta decisión se debe a varios factores, inicialmente las **dificultades en la compatibilidad de la máquina de desarrollo** de sistema MacOs con las librerías de Flex y Bison. Ambas **son accesibles desde el sistema a través de Brew**, un administrador de paquetes, pero los enlaces creados en el *path* y las diferentes versiones disponibles, junto con la versión del compilador de C++, **daban como resultado errores constantes e inestabilidad al desarrollo**. Por otro lado, la rigidez del desarrollo con Flex, Bison y C++ **daban un peso mayor al proyecto del que lo haría la propia implementación del compilador** adaptado a las necesidades y a la sencillez del lenguaje. De esta manera, de ser necesario incluir alguna característica poco común no soportada por el análisis de las librerías sería posible hacerlo de manera sencilla.

ANTLR por otro lado, a la hora de iniciar el desarrollo del proyecto, **no presentaba un soporte estable para otro lenguaje que no fuese Java**. Mantenía distintas opciones en desarrollo para otros lenguajes, pero **ninguna adecuada a el desarrollo de este proyecto** con suficiente documentación o flexibilidad. Debido principalmente a su poco tiempo de vida en comparación con los anteriores **fue descarta como opción**.

Finalmente la implementación del compilador se basó en el desarrollo de una **máquina SLR** tanto por su peso más ligero que CLR, como por su sencillez en comparación a LALR, GLR y LR, y a sus amplias posibilidades en la gramática sobresaliendo sobre LL.

La implementación del compilador se realizó en **el entorno Python utilizando las librerías básicas** para expresiones regulares, lectura y escritura de archivos. El sistema utiliza dos archivos JSON para obtener la especificación de los *tokens* y la sintaxis del lenguaje. El archivo de lexemas contiene un diccionario clave-valor, donde la clave es el nombre del *token*

y el valor es una cadena de texto que representa la expresión regular del mismo. Por otro lado, el archivo de sintaxis contiene las reglas de producción en forma de un diccionario donde la clave es la cabeza o parte izquierda y el valor es una lista de partes derechas acordes. Las partes derechas de estas reglas son a su vez una lista de cadenas de texto que representan los Terminales y No-Terminales de la gramática.

4.4.1 Lexer

El analizador léxico implementado **trabaja en conjunto con el analizador sintáctico** avanzando ambos al mismo tiempo. Como primer paso, el lexer se **inicializa con los valores del archivo de lexemas**, después se introduce el texto carácter a carácter y este comprueba la entrada con cada una de las expresiones regulares almacenadas. Si ninguna coincide, se avanza una posición de la entrada y se prueba de nuevo. Si alguna coincide se **almacena el posible resultado** y se avanza otra posición. Esto último es debido a que una palabra deseada **puede cumplir con una expresión regular en varios segmentos de su contenido**. Por ejemplo la entrada "abccc" coincide con la expresión regular "abc+" pero las primeras tres letras "abc" también lo cumplen. Para evitar esto utilizamos el principio "*Maximal munch*" en inglés, que implica **coger la coincidencia máxima** o en este caso **la más larga**. De esta manera aun que encontremos una coincidencia debemos de **seguir consumiendo hasta que pare** y así estar seguros de consumir lo máximo posible.

Los tokens, la clave de la regla que coincida, **son enviados al parser para su procesamiento** junto con el texto original a consumir.

4.4.2 Parser

El algoritmo de análisis sintáctico **es una variación más sencilla del algoritmo SLR**, su principio es el mismo, un método bottom-up LR para crear el árbol sintáctico acorde al código, utilizando los posibles tokens siguientes para decidir en el caso de conflicto. Sin embargo en esta implementación **la estructura de datos no es una tabla** si no que se utilizan las propias reglas y una pila de elementos. El proceso de ejecución es el siguiente:

Reglas de Producción:

- $S \rightarrow AB$
- $A \rightarrow \alpha\alpha\beta$
- $B \rightarrow C\gamma$
- $C \rightarrow \omega\omega$

Primero **se carga el contenido del archivo de sintaxis** con las **reglas de producción**, aquellas que comparten cabeza se separan y se crean como un par nuevo con una cabeza propia. De esta manera obtenemos una **lista de tokens y sus cabezas correspondientes de manera uniforme**. **La pila de tokens comienza vacía** y **la cadena de entrada en la posición cero**.

Cadena de entrada: $\alpha\ \alpha\ \beta\ \omega\ \omega\ \gamma$

Avance de la máquina:

Pila	Input	Resto de Input	Reducción
		$\alpha\ \alpha\ \beta\ \omega\ \omega\ \gamma$	
α	α	$\alpha\ \beta\ \omega\ \omega\ \gamma$	
$\alpha\ \alpha$	α	$\beta\ \omega\ \omega\ \gamma$	
$\alpha\ \alpha\ \beta$	β	$\omega\ \omega\ \gamma$	$A \rightarrow \alpha\ \alpha\ \beta$
A	β	$\omega\ \omega\ \gamma$	
A ω	ω	$\alpha\ \alpha\ \beta\ \omega\ \omega\ \gamma$	
A $\omega\ \omega$	ω	$\alpha\ \alpha\ \beta\ \omega\ \omega\ \gamma$	
A C	ω	$\alpha\ \alpha\ \beta\ \omega\ \omega\ \gamma$	$C \rightarrow \omega\ \omega$
A C γ	γ	$\alpha\ \alpha\ \beta\ \omega\ \omega\ \gamma$	
A B	γ	$\alpha\ \alpha\ \beta\ \omega\ \omega\ \gamma$	$B \rightarrow C\ \gamma$
S	γ	$\alpha\ \alpha\ \beta\ \omega\ \omega\ \gamma$	$S \rightarrow A\ B$

Tabla 4.1: Tabla de avance en la máquina implementada.

Según se **alimenta con la entrada de texto al analizador léxico** este **devuelve los tokens** que ha detectado a el analizador sintáctico. Al recibirse un token este **se añade a la cabeza de la pila**. Utilizando las reglas de producción **se comprueba desde la cabeza hasta el final** si coincide alguna combinación de los elementos con una **regla para reducir**, en caso contrario se avanza otro token.

Puede existir el caso de **dos reglas de producción** en la cual **una esté incluida en otra**, por ejemplo las reglas:

- $A \rightarrow \alpha\ \alpha\ \beta$
- $A \rightarrow \alpha\ \alpha\ \beta\ \beta\ \gamma$

Con solo esta aproximación nunca se alcanzaría la segunda regla, por ello se **añade una comprobación** a la utilizada con la regla. No solo se comprueba si coinciden exactamente con

los tokens, además **se comprueban si están incluidos en el principio de otra regla**. De esta manera si se alcanza a tener un **conjunto de elementos en la pila que coincide con una regla y parte de otra** se opta por **ver el próximo token de entrada para decidir**. Si el Token se encuentra en el conjunto de elementos siguientes, se mantiene, en otro caso se reduce.

Cada vez que se reducen los elementos a una cabeza se componen los elementos del cuerpo como nodos hijos del nuevo elemento cabeza, componiendo de esta manera el árbol sintáctico como representación del código.

4.4.3 Procesado y Optimizaciones

Una vez se completa el árbol sintáctico, empieza el procesado y optimización del código. En esta implementación del código, el **procesado es muy sencillo** y las optimizaciones son pocas debido a la alta carga de trabajo que supondría en el proyecto. El procesado se divide en **tres segmentos** tras la creación del árbol sintáctico (como muestra la Figura 4.11), el primero consta de una comprobación de **tipos**, **argumentos** y **funciones**, el segundo una optimización para aquellas funciones que se beneficien de la **recursividad terminal** y finalmente una traducción a el *Byte-code* del lenguaje.

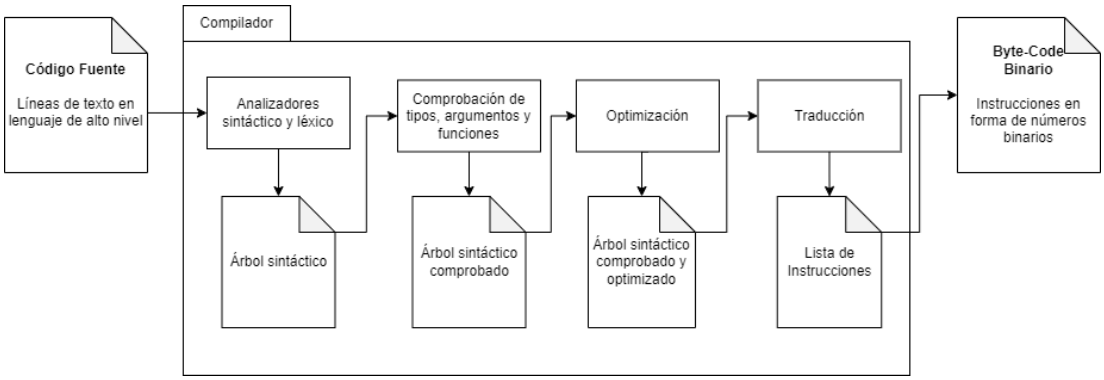


Figura 4.11: Esquema con los distintos procesos dentro del compilador.

La comprobación de tipos se realiza recorriendo el **árbol abstracto**. Por cada nodo correspondiente a una función se apunta el tipo correspondiente de sus argumentos y tipo devuelto como resultado. De la misma manera, por cada variable se apunta el tipo acorde a esta y se procesan las expresiones para obtener el tipo de salida. Si en alguna operación los tipos de los argumentos no son adecuados el sistema **lanza un error** como en la figura 4.12a, por otro lado, si el árbol abstracto no conforma una sola expresión también se **lanza un error** como el de la figura 4.12b.

Cada vez que se declara una variable es necesario reservar una sección de memoria para ella, usualmente se utilizan los **registros disponibles**. El compilador lleva un recuento de los

(a) Error lanzado al no coincidir los tipos.

(b) Error lanzado al no poder formar un árbol sintáctico correcto.

Figura 4.12: Ejemplo errores lanzados por el compilador.

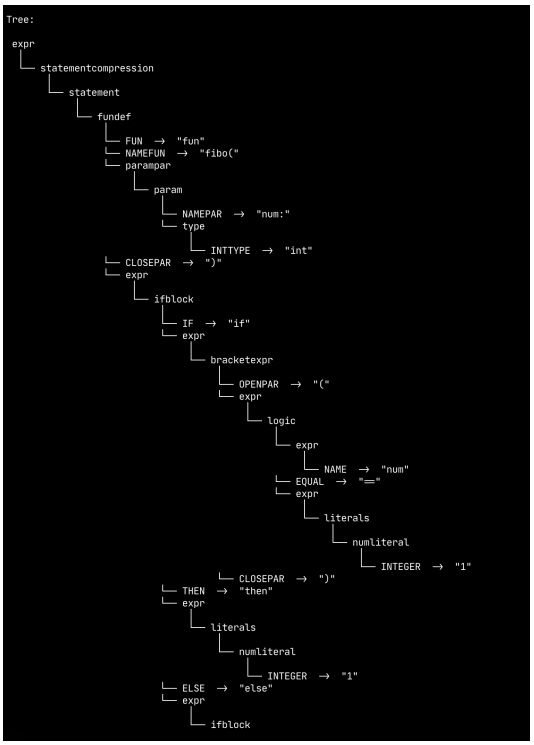
registros utilizados y la variable que representan en cada momento, de esta manera si se crea una variable se le asigna un registro concreto y si es necesario leer o escribir sobre el se utiliza ese registro. Al alcanzar el final de la función los valores antiguos de los registros son **borrados** y el estado de los registros antes de la llamada es **restaurado**.

Para poder **guardar y restaurar** el estado de los registros antes y después de cada llamada a una función se utiliza el **marco de función** (en inglés *stack frame*). Existen **tres tipos de memoria** que utilizaremos, la pila, el montículo y los registros. Por defecto en la configuración de la máquina virtual de TedLang existen 64 registros y una pila de 8MB y un montículo de 32MB. Los registros corresponden con la sección de memoria a la que accederemos para los valores dentro de el alcance de la función actual. La pila es la encargada de **almacenar los marcos de función** de cada llamada y el montículo es una sección de la memoria en la que tenemos un mayor control, pudiendo acceder a **direcciones exactas** para leer y escribir valores. La principal diferencia entre estas secciones es la **velocidad de acceso**, los registros son los **más rápidos** pudiendo acceder a ellos directamente, la pila contiene valores como los argumentos al inicio de cada función y es posible acceder pero se debe calcular su posición dentro de el marco de la función, por último a el montículo se accede por direcciones y referencias por lo que para una posición es necesario realizar varios cálculos y accesos distintos.

Una vez se han realizado todas las comprobaciones se inicia el proceso de **traducción a instrucciones propias** en el cual se transforma un árbol sintáctico como el de la Figura 4.13a en un *set* de instrucciones como el de la Figura 4.13b.

Por cada expresión se separa en sus pasos y estos en otros más pequeños hasta que se convierten en pasos representables por una instrucción. Esta implementación exige una instrucción para guardar en la pila cada uno de los argumentos y recuperarlos a la vuelta por las variaciones que puedan sufrir durante la función y para evitar los efectos colaterales directos. Lo cual significa que por cada llamada a una función aun que el marco de función se maneje dentro de la máquina virtual debemos incluir una instrucción de empujar en la cabeza de la pila anterior a la llamada y una instrucción de retirar de la pila al volver.

Entraremos en el detalle del uso de las instrucciones en el apartado dedicado a ellas más adelante, durante la subsección 4.4.4.



(a) Árbol sintáctico correspondiente al código.

```
   id                instruccions

GLOBALES:
0                 ('MOV', ('NULL', 0), 24)
1                 ('MOV', ('TRUE', 0), 25)
2                 ('MOV', ('FALSE', 0), 26)
3                 ('MOV', ('_', 0), 27)
4                 ('MOV', ('5', 5), 28)
5                 ('MOV', ('0', 0), 29)
6                 ('MOV', ('1', 1), 30)
7                 ('MOV', ('31', 0), 31)

main call
8                 ('J', 68)

functions

9                 ('MOV', 0, 24)
10                ('MOV', 1, 25)
11                ('MOV', 0, 26)
12                ('MOV', 0, 27)
13                ('MOV', 0, 28)
14                ('MOV', 0, 29)
15                ('MOV', 0, 30)
16                ('MOV', 2, 31)
17                ('MOV', 0, 32)
18                ('MOV', 0, 33)
19                ('MOV', 0, 34)
20                ('MOV', 0, 35)
21                ('MOV', 0, 36)
22                ('ARG', 0, 24)
23                ('EQ', 24, 25, 26)
24                ('JIZ', 26, 2)
25                ('ADDI', 0, 25, 27)
26                ('J', 17)
27                ('EQ', 24, 28, 29)
28                ('JIZ', 29, 2)
29                ('ADDI', 0, 28, 30)
30                ('J', 12)
31                ('SUB', 24, 31, 32)
32                ('PUSH', 32)
33                ('CALI', 8)
34                ('POP', 32)
35                ('ADDI', 0, 3, 33)
36                ('SUB', 24, 25, 34)
37                ('PUSH', 34)
38                ('CALI', 8)
39                ('POP', 34)
40                ('ADDI', 0, 3, 35)
41                ('ADD', 33, 35, 36)
42                ('ADDI', 0, 36, 30)
43                ('ADDI', 0, 30, 27)
44                ('RET', 27)

45                ('MOV', 0, 24)
46                ('MOV', 0, 25)
47                ('MOV', 0, 26)
48                ('MOV', 0, 27)
49                ('MOV', 0, 28)
50                ('MOV', 0, 29)
51                ('MOV', 1, 30)
52                ('MOV', 0, 31)
```

(b) Instrucciones resultantes de la traducción.

Figura 4.13: Productos intermedios producidos durante la compilación.

También como última optimización a la vez que se traducen las expresiones a instrucciones en el caso de encontrar una **función** que se pueda beneficiar de **recursividad terminal**, se substituye la instrucción de llamada a la función por una de **salto al inicio de la función**. Además se utilizan las instrucciones de escritura de argumentos para **actualizar** los correspondientes de la nueva llamada.

4.4.4 Instrucciones intermedias

Como se ha manifestado anteriormente en este documento, las máquinas virtuales pueden ser una opción idónea para crear un sistema **altamente portable**. Con la propuesta inicial del lenguaje en mente, la sencillez, **se tomó la decisión de basar su desarrollo en una de manera que fuera accesible para un público muy amplio**. Para esta máquina virtual se diseñaron una serie de Instrucciones escritas en forma de *Byte-code*, basadas en las instrucciones del microcontrolador *MIPS*. Debido a anteriores proyectos dentro de los estudios el *set* de instrucciones del microcontrolador **eran muy familiares** y se adaptaron para esta nueva máquina virtual.

Algunos ensambladores y compiladores **añaden una capa de pseudo-instrucciones**, estas son instrucciones que **se ven como una sola pero se escriben como un conjunto de ellas**. Por ejemplo se puede crear una pseudo-instrucción que inicialice un registro a cero con la instrucción ANDI y un valor cero, dado que cualquier función ”AND” contra cero dará como resultado cero. Dado que las instrucciones **deben codificarse como un número**, solo puede haber una **cantidad limitada de ellas**, las pseudo-instrucciones utilizarían las que ya existen para crear nuevas instrucciones y aprovecharían al máximo el espacio de las instrucciones.

Byte-Code	Nombre	Argumentos	Descripción
00	NOUP		Operación vacía
01	ADD	n m s	Reg[s] = Reg[n] + Reg[m]
...	...	...	...

Tabla 4.2: Tabla de instrucciones.

La lista completa de instrucciones y su definición están incluidas en el Apéndice A.2 y A.4.

Instrucción No-Operation (NOP)

La instrucción NOP representa una **operación vacía**. Esto sienta los cimientos de una implementación del compilador en el que para **optimizar el procesado de instrucciones** se rellenan los espacios entre ellas con NOP de manera que **todas midan lo mismo**. Además, como se mostrará en el capítulo de líneas futuras, en caso de implementar un sistema **multi-hilo**

es posible utilizar estas instrucciones para crear un **sistema segmentado virtual** o repartir espacio entre los procesos de diferentes hilos.

Instrucción Move (MOV)

La instrucción MOV se utiliza para **guardar un valor literal en un registro**. La inicialización de valores en los registros es algo **muy habitual en la programación**, tanto las **constantes globales** que nunca varían como las **funciones que son llamadas constantemente** y deben utilizarse con una configuración exacta. Esta instrucción se podría implementar como una pseudo-instrucción que multiplique por cero y sume el valor u otras combinaciones, pero el **tamaño y el tiempo de proceso del programa** en este caso **aumentarían innecesariamente**.

Instrucción Debug (DBG)

La instrucción DBG **para el proceso en ejecución** y muestra por la salida principal el **estado de ejecución**, el programa cargado en memoria, el puntero a la instrucción, la pila, los registros y la entrada y salida virtual. Los datos se muestran por la salida principal por defecto como se ven en las Figuras 4.15 y 4.16 sobre la ejecución de la función en la Figura 4.14.

```
fun fibo(num:int) int:
    if ( num == 1) then
        1
    else if ( num == 0 ) then
        0
    else
        fibo(num-2) + fibo(num-1)
endfun;
fun fiborec(num:int,a:int,b:int) int:
    if ( num == 0) then
        a
    else if ( num == 1 ) then
        debug();
        b
    else
        fiborec((num-1),b,a+b)
endfun;
print(fiborec(5,0,1));
0
```

Figura 4.14: Ejemplo de una función en Elixir.

Instrucciones de operaciones: ADDI SUBI MULTI DIVI

Las instrucciones ADDI SUBI MULTI DIVI y similares son las encargadas de realizar **operaciones entre dos valores y almacenar el resultado en un registro**. Su característica

```
        ( 35 )          ADDI (0,3,33)
        ( 36 )          SUB (24,25,34)
        ( 37 )          PUSH (34,0,0)
        ( 38 )          CALI (8,0,0)
        ( 39 )          POP (34,0,0)
        ( 40 )          ADDI (0,3,35)
        ( 41 )          ADD (33,35,36)
        ( 42 )          ADDI (0,36,30)
        ( 43 )          ADDI (0,30,27)
        ( 44 )          RET (27,0,0)
        ( 45 )          MOV (0,24,0)
        ( 46 )          MOV (0,25,0)
        ( 47 )          MOV (0,26,0)
        ( 48 )          MOV (0,27,0)
        ( 49 )          MOV (0,28,0)
        ( 50 )          MOV (0,29,0)
        ( 51 )          MOV (1,30,0)
        ( 52 )          MOV (0,31,0)
        ( 53 )          MOV (0,32,0)
        ( 54 )          MOV (0,33,0)
        ( 55 )          MOV (0,34,0)
        ( 56 )          ARG (0,24,0)
        ( 57 )          ARG (1,25,0)
        ( 58 )          ARG (2,26,0)
        ( 59 )          EQ (24,27,28)
        ( 60 )          JIZ (28,2,0)
        ( 61 )          ADDI (0,25,29)
        ( 62 )          J (13,0,0)
        ( 63 )          EQ (24,30,31)
        ( 64 )          JIZ (31,3,0)
IP →    ( 65 )          DBG (0,0,0)
        ( 66 )          ADDI (0,26,32)
        ( 67 )          J (7,0,0)
        ( 68 )          SUB (24,30,33)
        ( 69 )          ADD (25,26,34)
```

(a) Log mostrando el programa y el puntero a la instrucción actual en una ejecución.

```
-----------------------------------------
REG init:       24
VInput init:    4
VOutput init:   14
-------
CPU: Registers.
( SP ) 69       [45]
( IP ) 65       [41]
( FP ) 69       [45]
( ACC ) 0       [0]
( FS ) 0        [0]
-------
( R0 ) 65       [41]
( R1 ) 69       [45]
( R2 ) 69       [45]
( R3 ) 0        [0]
( R4 ) 0        [0]
( R5 ) 10       [a]
( R6 ) 0        [0]
( R7 ) -1237387904      [b63ef580]
( R8 ) 32759    [7ff7]
( R9 ) 260045962        [f7ffc8a]
( R10 ) 32760   [7ff8]
( R11 ) 0       [0]
( R12 ) 0       [0]
( R13 ) 1359440280      [51076998]
( R14 ) 32760   [7ff8]
( R15 ) 0       [0]
( R16 ) 0       [0]
( R17 ) 375803920       [16665010]
( R18 ) 1       [1]
( R19 ) -1237387856     [b63ef5b0]
( R20 ) 32759   [7ff7]
( R21 ) 260045962       [f7ffc8a]
( R22 ) 32760   [7ff8]
( R23 ) 1359440280      [51076998]
( R24 ) 1       [1]
( R25 ) 3       [3]
( R26 ) 5       [5]
( R27 ) 0       [0]
( R28 ) 0       [0]
( R29 ) 0       [0]
( R30 ) 1       [1]
( R31 ) 1       [1]
( R32 ) 0       [0]
( R33 ) 0       [0]
( R34 ) 0       [0]
( R35 ) 32760   [7ff8]
```

(b) Log mostrando los registros y sus valores en una ejecución.

Figura 4.15: Log mostrando datos del programa y registros.

```
CPU: Memory (3b to 8a).
        (59) b63ef560
        (60) 7ff7
        (61) f7ffb57
        (62) 7ff8
        (63) 0
        (64) 0
        (65) 51076998
        (66) 7ff8
        (67) 50
        (68) 45
 SP →   (69) 0  ← FP
        (70) 0
        (71) 0
        (72) 0
        (73) 0
        (74) 0
        (75) 0
        (76) 0
        (77) 0
        (78) 0
        (79) 0
        (80) 0
        (81) 0
        (82) 0
        (83) 0
        (84) 0
        (85) 0
        (86) 0
```

(a) Log mostrando el valor de la pila y su posición en una ejecución.

```
CPU:            Input                                   Output
(       0 )     0[0]                    32760[7ff8]
(       1 )     10[a]                   0[0]
(       2 )     0[0]                    0[0]
(       3 )     -1237387904[b63ef580]                   375803920[16665010]
(       4 )     32759[7ff7]                     1[1]
(       5 )     260045962[f7ffc8a]                      -1237387856[b63ef5b0]
(       6 )     32760[7ff8]                     32759[7ff7]
(       7 )     0[0]                    260045962[f7ffc8a]
(       8 )     0[0]                    32760[7ff8]
(       9 )     1359440280[51076998]                    1359440280[51076998]
```

(b) Log mostrando los valores de las entradas y salidas virtuales en una ejecución.

Figura 4.16: Log mostrando datos de la pila y las entradas y salidas virtuales.

común es que uno de esos valores **proviene de un registro** y el otro es **obtenido de la propia instrucción**, este último se llama **valor inmediato**.

Por ejemplo la instrucción ”ADDI 7 6 8” guarda en el registro número 8 el resultado de sumar 7 más el valor en el registro 6.

Instrucciones de operaciones: ADD SUB MULT DIV

Las instrucciones ADD SUB MULT DIV y similares sirven para la misma función que sus contra-partes inmediatas pero en su caso **ambos valores provienen de un registro**.

Utilizando el mismo ejemplo de la instrucción ADDI con ”ADD 7 6 8” guarda en el registro número 8 el resultado de sumar el valor del registro 7 más el valor en el registro 6.

Instrucciones de memoria: LW SW LWI SWI

Las instrucciones LW SW LWI SWI permiten **almacenar y cargar datos del montículo** con una dirección de referencia obtenida del registro utilizado como valor. Esto nos permite un mejor manejo de memoria a gran escala con el **coste de un mayor tiempo de acceso**.

Instrucciones de Pila: PUSH POP

Las instrucciones PUSH y POP son las encargadas de **añadir y eliminar elementos de la cabeza de la pila**.

Instrucciones de salto: J JEQ JNE JI JR JIZ

Las instrucciones J JEQ JNE JI JR JIZ y similares son las encargadas de la administración y **realización de los saltos en las instrucciones**. Debido a que **estas instrucciones son tan habituales en el código** sirviendo como traducción de bloques if y otras comprobaciones, las distintas variedad cubren casos muy concretos para **reducir al mínimo los pasos extra** y mejorar la eficiencia en tiempos de ejecución. Por ejemplo la función JIZ (Jump If Zero) comprueba que un registro sea cero y si lo es realiza un salto. Esto podría realizarse con un JEQ (Jump if EQuals) en el que uno de los valores es un registro con el valor cero, pero esto implicaría mantener un registro extra con el valor cero en uso y obtener su valor a la hora de comprobar el salto.

Instrucciones de funciones: CAL CALI RET ARG SARG

Estas instrucciones se utilizan en común para la **llamada de funciones y operaciones relacionadas**. Las funciones CAL y CALI realizan la **llamada a la función**, iniciando el proceso de **guardado del marco de la función**. La instrucción RET marca el **final de una función** y el valor que la acompaña es el registro que **sirve de resultado de la función**,

además llama al proceso de **restauración del anterior marco de la función**. La instrucción ARG **obtiene el argumento de la función actual** en la posición indicada. La instrucción SARG **actualiza el valor del argumento en la posición indicada**, esto es realizado al hacer un salto en lugar de una llamada recursiva para poder implementar la **recursividad terminal**.

Instrucción I/O: INP OUT

Las instrucciones INP y OUT reciben un único valor, el número de salida a utilizar. Al procesar esta instrucción el sistema **muestra o almacena**, en la entrada o salida correspondiente al valor que la acompaña, **el valor en su sección de memoria**.

Por defecto la **salida número cero corresponde con el terminal de ejecución actual** y la **entrada número cero con el teclado** de dicho terminal.

4.5 Máquina virtual

La máquina virtual de TedLang **contiene varias piezas intercomunicadas** que manejan los diferentes aspectos de su funcionamiento, desde el **hilo de ejecución** de instrucciones hasta la **comunicación con dispositivos externos** o **manejo de memoria interna**. En esta sección nos centraremos en la **descripción de su funcionamiento e implementación**, prestando especial atención a aquellos elementos más importantes.

Como se puede apreciar en el apartado 4.4.3 sobre el compilador interno, este **debe tener en cuenta el número de registros disponibles y totales**, guardando en la pila el estado de los mismos **antes de cada función** y **cargándolos después**. Esto ocurre en las **máquinas basadas en registros**, pequeñas secciones de memoria accesibles por la CPU para leer y escribir **de manera ágil**. Existen otro tipo de máquinas, **basadas puramente en la pila de memoria**, que **añaden y retiran de la pila de memoria** continuamente para poder acceder a la memoria. Aun que estos modelos son **muy eficientes en el uso de la memoria**, la constante inserción y eliminación de elementos en la pila **ralentiza mucho su funcionamiento**.

4.5.1 Estructura Interna

Ensamblador

El Ensamblador es el encargado de **traducir**, dado una cadena en binario, en **operaciones comprensibles por la máquina virtual** y cargarlas en memoria de manera ordenada. Por cada byte de entrada debe analizar el **tipo de instrucción acorde** y el **número de valores que la acompañan**. El tamaño de las instrucciones **puede variar dependiendo de su función**, pueden necesitar un valor como la J ("*Jump*" y su valor de salto), dos valores como la JIZ ("*Jump If Zero*" el valor de salto y el registro a comprobar), o tres valores como JNE ("*Jump If Not*

Equal" con el valor de salto y los dos registros a comparar). Una opción seria almacenar en el archivo *Byte-Code* las instrucciones con espacios en blanco en la posición que no utilizan pero para ello todas ocuparían el máximo de espacio, tres posiciones. Debido a que gran cantidad de instrucciones tienen uno o dos valores, **el tamaño del binario resultante aumentaría mucho**.

Una vez el programa se ha cargado en memoria, **es enviado a la CPU** para que empiece su ejecución.

CPU

La CPU **procesa y ejecuta cada una de las instrucciones** mientras no ocurra un error en su ejecución, de ser así, se **pararía mostrando un texto con el error acorde**. Cada una de las instrucciones realiza una **acción correspondiente**, manejando la memoria, la entrada y salida o realizando operaciones entre registros. Estas instrucciones realizan diferentes operaciones tanto matemáticas como lógicas y el resultado **se almacena a través del manejador de memoria** (o como se llama en inglés *Memory Manager*) en la posición de destino.

Memory Manager

El *Memory Manager* es el **encargado del manejo de memoria**, tanto guardar los resultados de otras operaciones, como acceder a ellos o acceder a los valores de los distintos punteros necesarios para la ejecución. Estos son: el puntero a la **instrucción actual** IP (de sus siglas en inglés Instruction Pointer), el puntero a la **posición en la pila** SP (de sus siglas en inglés Stack Pointer)y el puntero a la **posición del último marco de función** FP (de sus siglas en inglés Frame pointer). Además existen una serie de **registros especiales como el acumulador** en el que se guardan los resultados de la última función, llamado ACC, y las diferentes secciones de memoria anteriormente mencionadas como **la pila** o **el montículo**.

El *Memory Manager* es el **encargado de guardar y cargar el marco de la función**, este marco está compuesto por el valor de **todos los registros**, el valor del **puntero a la instrucción** y el valor del **tamaño del marco**. De esta manera si es necesario acceder a alguno de ellos podemos saber su posición en la pila dado el tamaño y el orden en el que los hemos guardado. De la misma manera **restaurarlos** es una tarea sencilla, tan solo hay que **obtenerlos de la cabeza de la pila en el mismo orden**.

Virtual I/O

La entrada y salida virtual manejan el **uso de diferentes elementos para obtener o devolver datos**. Cada entrada y salida **corresponde con una sección de memoria y un método de ejecución**, por defecto la salida cero es la salida del terminal en ejecución y la

entrada cero es el teclado de ese mismo terminal. El número de entradas y salidas disponibles **solo se puede variar antes de lanzar la máquina virtual** y es necesario cambiar el código que las maneja para añadir métodos nuevos de lectura o escritura de estas secciones. En el trabajo futuro profundizaremos sobre las posibilidades que tiene este elemento y los cambios necesarios para ello.

4.5.2 Inicialización y Proceso general

Una vez arrancada la máquina se **inicializa las secciones de memoria de entrada y salida junto con el resto** gracias al *Memory Manager*, se **inicializa la CPU y el ensamblador**. Este lee el programa y **lo transforma en una serie instrucciones comprensible**, para **devolverlas** a la CPU. La CPU comienza la **ejecución de las instrucciones**, avanzando paso a paso por cada una de ellas.

El programa en Byte-code para una máquina TedLang tiene una **estructura precisa**, en primera posición está la **inicialización de variables**. Después de estos se encuentra la **instrucción de salto hasta el punto de inicio del programa**, y entre este punto y la instrucción de salto están el resto de **funciones del programa**.

Una vez alcanzado el final del programa se libera la memoria reservada por el *Memory Manager* y acaba la ejecución.

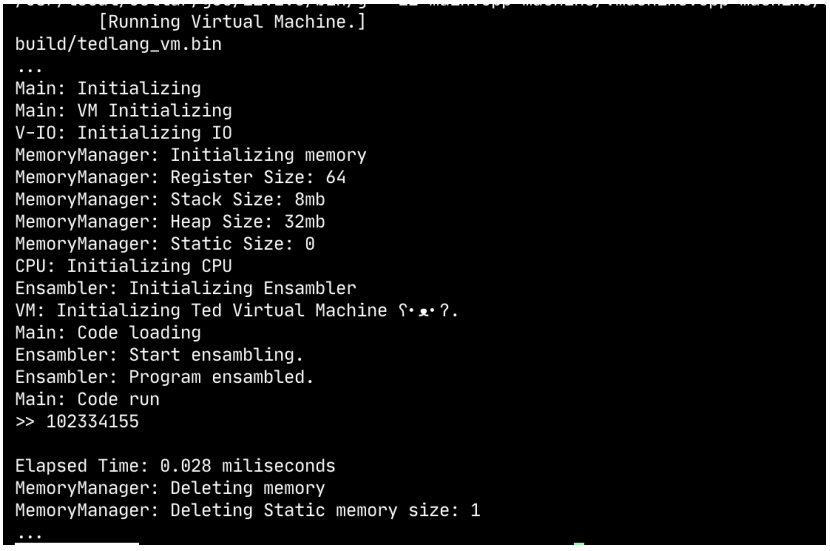


Figura 4.17: Captura de los datos básicos ofrecidos por la máquina virtual.

4.5.3 Compilación cruzada

La compilación cruzada consiste en obtener código ejecutable por otra máquina distinta a aquella en la que se está compilando. Para este proyecto se necesita desarrollar un sistema de

compilación cruzada con las máquinas JVM y BEAM que transforme el código desde TedLang a el Byte-Code de Java o la estructura de objetos de BEAM.

Capítulo 5

Conclusiones

En el presente capítulo se presentarán las conclusiones del proyecto, incluyendo las lecciones aprendidas durante la realización del mismo, una revisión de los objetivos propuestos inicialmente y su seguimiento a lo largo del desarrollo, y una serie de propuestas para el trabajo futuro que ampliarían los horizontes del proyecto.

5.1 Objetivos alcanzados y lecciones aprendidas

El desarrollo de este proyecto ha tenido un gran componente de búsqueda e investigación sobre los procesos necesarios en la compilación, mostrando la gran importancia de este elemento en el proyecto. La mayoría de elementos que se implementen en la máquina en relación al lenguaje deberán ser adecuadamente detectados y adaptados a instrucciones. Esto implica un nivel de comprensión muy alto por parte del compilador, no solo sintácticamente, las referencias al propio código y la relación entre secciones del código o instrucciones entre sí son clave. La optimización del código también exige una comprensión total del funcionamiento de la máquina virtual y el programa a compilar.

Respecto a los objetivos propuestos ya nombrados en la sección 1.1:

- Desarrollo de un **lenguaje de programación funcional** con las siguientes características:
  - **Sintaxis** donde se priorice la **simplicidad**.
  - **Capacidad de ejecutar el código de manera paralela** y tener **comunicación entre distintos hilos de mensajes**.
  - Gestionar la **existencia de clases** y la **herencia**, con el objetivo de realizar factorización y reutilización de código.
- **Implementación de un compilador** que transforme el código fuente en el lenguaje deseado en un **código intermedio, *bytecode,* de instrucciones propias**, con la posibilidad

de **exportarlo** a otros **códigos intermedios compatibles** con la **máquina virtual de Java (JVM)** o con la **máquina virtual de Erlang (BEAM)**.

- **Implementación de una máquina virtual** para la ejecución del *bytecode* de instrucciones propias.

El desarrollo del lenguaje funcional fue un éxito parcial teniendo en cuenta que el lenguaje fue desarrollado e implementado a excepción de la comunicación entre hilos a través de mensajes. Al lenguaje lo define una sintaxis simple y funcional dirigida principalmente por expresiones, con posibilidad de utilizar las clases en ella para su organización.

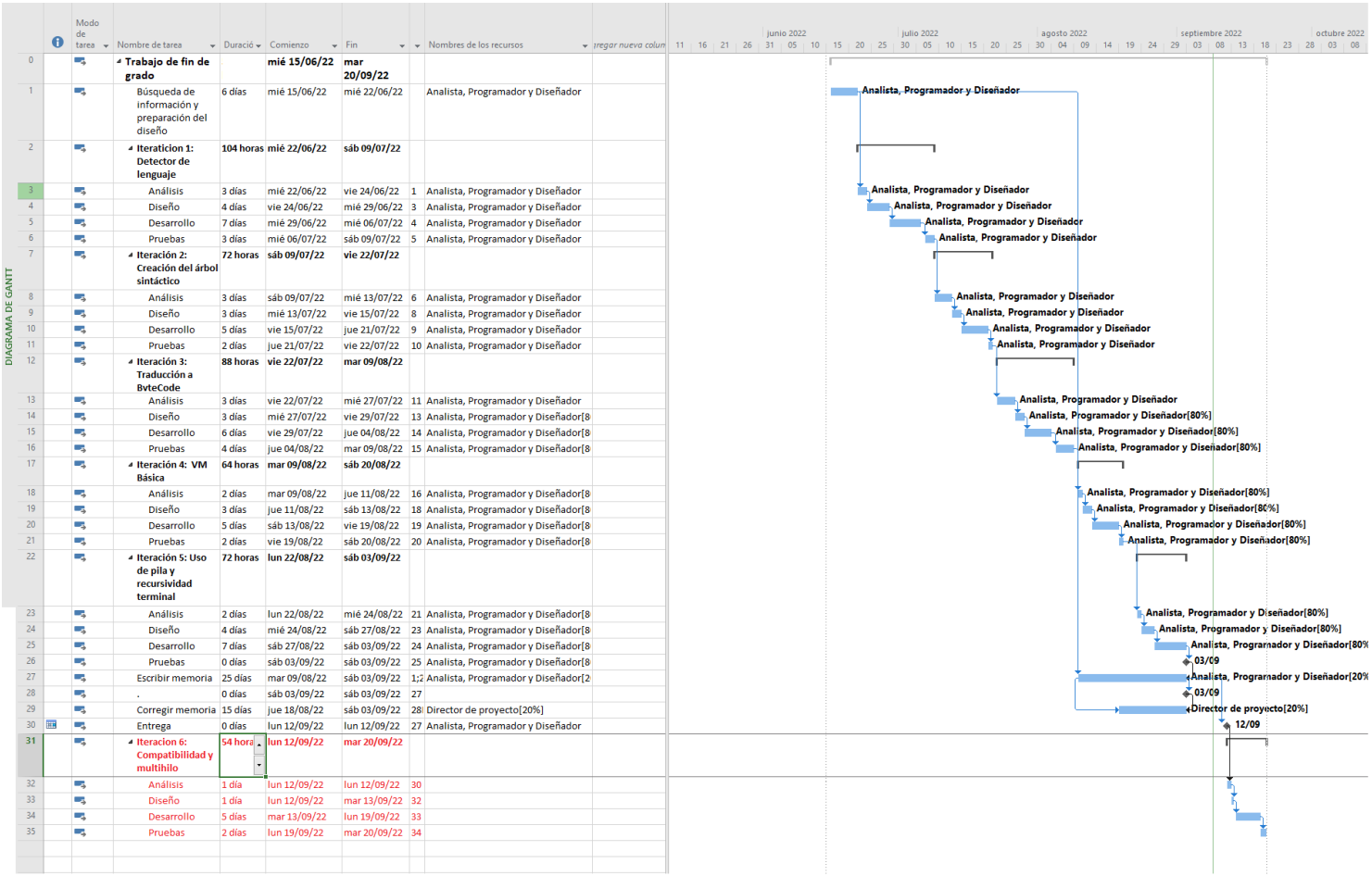
El compilador fue desarrollado de manera que detecta la sintaxis adecuada y la transforma en las instrucciones *Byte-Code* propias. Lamentablemente debido a improvistos en el desarrollo que conllevaron trabajar más tiempo del estimado en iteraciones anteriores, y al formar parte de la última iteración, no fue posible añadir la capa de compatibilidad con las máquinas JVM y BEAM.

Por último, la máquina virtual fue implementada de la manera planeada con todos los elementos (a excepción del envío de mensajes multihilo ya mencionado).

5.2 Seguimiento del proyecto

Debido a que este trabajo se ha encontrado con problemas sobre todo en las etapas de pruebas y que hubo que solucionarlos en sus diversas iteraciones, el diagrama de Gantt inicial de la figura 3.2, presentado en el Capítulo 3, no coincide en su totalidad con lo que realmente ha sucedido. Por ello, para realizar un seguimiento más realista de este proyecto se puede ver cómo se ha realizado la replanificación de las tareas en el diagrama de la figura 5.1.

En cuanto al coste, aunque que se habían intentado sobrestimar algunas tareas con el fin de tener holgura y contar con los posibles imprevistos, el precio total ha ascendido a 8.575€. Esto se debe a que se ha subestimado el tiempo de ciertas tareas realizadas por el rol de “analista, diseñador y programador”, el cual finalmente, se ha presupuestado en 8.035€ realizando 32h más de las previstas e impidiendo la realización de la última iteración considerada.



	Modo de tarea	Nombre de tarea	Duració	Comienzo	Fin		Nombres de los recursos
0		**Trabajo de fin de grado**		**mié 15/06/22**	**mar 20/09/22**		
1		Búsqueda de información y preparación del diseño	6 días	mié 15/06/22	mié 22/06/22		Analista, Programador y Diseñador
2		**Iteraticion 1: Detector de lenguaje**	**104 horas**	**mié 22/06/22**	**sáb 09/07/22**		
3		Análisis	3 días	mié 22/06/22	vie 24/06/22	1	Analista, Programador y Diseñador
4		Diseño	4 días	vie 24/06/22	mié 29/06/22	3	Analista, Programador y Diseñador
5		Desarrollo	7 días	mié 29/06/22	mié 06/07/22	4	Analista, Programador y Diseñador
6		Pruebas	3 días	mié 06/07/22	sáb 09/07/22	5	Analista, Programador y Diseñador
7		**Iteración 2: Creación del árbol sintáctico**	**72 horas**	**sáb 09/07/22**	**vie 22/07/22**		
8		Análisis	3 días	sáb 09/07/22	mié 13/07/22	6	Analista, Programador y Diseñador
9		Diseño	3 días	mié 13/07/22	vie 15/07/22	8	Analista, Programador y Diseñador
10		Desarrollo	5 días	vie 15/07/22	jue 21/07/22	9	Analista, Programador y Diseñador
11		Pruebas	2 días	jue 21/07/22	vie 22/07/22	10	Analista, Programador y Diseñador
12		**Iteración 3: Traducción a BvteCode**	**88 horas**	**vie 22/07/22**	**mar 09/08/22**		
13		Análisis	3 días	vie 22/07/22	mié 27/07/22	11	Analista, Programador y Diseñador
14		Diseño	3 días	mié 27/07/22	vie 29/07/22	13	Analista, Programador y Diseñador[8
15		Desarrollo	6 días	vie 29/07/22	jue 04/08/22	14	Analista, Programador y Diseñador[8
16		Pruebas	4 días	jue 04/08/22	mar 09/08/22	15	Analista, Programador y Diseñador[8
17		**Iteración 4: VM Básica**	**64 horas**	**mar 09/08/22**	**sáb 20/08/22**		
18		Análisis	2 días	mar 09/08/22	jue 11/08/22	16	Analista, Programador y Diseñador[8
19		Diseño	3 días	jue 11/08/22	sáb 13/08/22	18	Analista, Programador y Diseñador[8
20		Desarrollo	5 días	sáb 13/08/22	vie 19/08/22	19	Analista, Programador y Diseñador[8
21		Pruebas	2 días	vie 19/08/22	sáb 20/08/22	20	Analista, Programador y Diseñador[8
22		**Iteración 5: Uso de pila y recursividad terminal**	**72 horas**	**lun 22/08/22**	**sáb 03/09/22**		
23		Análisis	2 días	lun 22/08/22	mié 24/08/22	21	Analista, Programador y Diseñador[8
24		Diseño	4 días	mié 24/08/22	sáb 27/08/22	23	Analista, Programador y Diseñador[8
25		Desarrollo	7 días	sáb 27/08/22	sáb 03/09/22	24	Analista, Programador y Diseñador[8
26		Pruebas	0 días	sáb 03/09/22	sáb 03/09/22	25	Analista, Programador y Diseñador[8
27		Escribir memoria	25 días	mar 09/08/22	sáb 03/09/22	1;2	Analista, Programador y Diseñador[2
28		.	0 días	sáb 03/09/22	sáb 03/09/22	27	
29		Corregir memoria	15 días	jue 18/08/22	sáb 03/09/22	28	Director de proyecto[20%]
30		Entrega	0 días	lun 12/09/22	lun 12/09/22	27	Analista, Programador y Diseñador
31		**Iteracion 6: Compatibilidad y multihilo**	**54 hora**	**lun 12/09/22**	**mar 20/09/22**		
32		Análisis	1 día	lun 12/09/22	lun 12/09/22	30	
33		Diseño	1 día	lun 12/09/22	mar 13/09/22	32	
34		Desarrollo	5 días	mar 13/09/22	lun 19/09/22	33	
35		Pruebas	2 días	lun 19/09/22	mar 20/09/22	34	

Figura 5.1: Diagrama de Gantt final.

5.3 Trabajo futuro

Mediante desarrollo de este proyecto se ha conseguido desarrollar una parte del lenguaje, pero tiene **muchas posibilidades** en su desarrollo futuro. En este capítulo comentaremos que elementos son mejorables de cara al futuro y como podrían plantearse esos cambios para alcanzar un **lenguaje más eficiente y versátil**.

5.3.1 Diseño del Lenguaje

El diseño del lenguaje ha sido **limitado** por el tiempo disponible para su desarrollo e implementación. En el futuro se podrían extender las opciones del lenguaje añadiendo otros **operadores**, **comprobaciones** y **"azúcar sintáctico"**.

Entre estas nuevas comprobaciones sería ideal implementar el característico *Pattern Matching* de otros **lenguajes funcionales**. A través de esta nueva comprobación sería posible **simplificar las funciones** que ahora mismo dependen de bloques condicionales "if" anidados. Estos bloques pueden volverse **grandes y confusos** en comprobaciones complejas dentro de la función. Para una funcionalidad plena de esta característica sería necesario implementar la **desestructuración** de variables para una detección de patrones complejos. Sería necesario un **análisis más complejo** que diferencie los patrones de una sección de expresiones pero el beneficio sería muy bueno para los desarrolladores/as.

5.3.2 Compilador

El compilador es el encargado de realizar la transformación del código en las instrucciones *Byte-Code* por ello es el apartado con **más posibilidades** para futuro. Los tipos implementados inicialmente son un **grupo reducido** que, aun que útil para las pruebas y demostraciones, podrían ser **extendidos** añadiendo aquellos más utilizados por los desarrolladores/as. Entre ellos están **listas, diccionarios, átomos o enumerados**. El lenguaje se vería **altamente enriquecido** al poder adoptar nuevas estructuras en su código. Las operaciones entre varios de ellos **pueden ser implementadas** con las instrucciones **actuales**. Por ejemplo los **enumerados** pueden implementarse como un par identificador-valor, en el cual el identificador sirve como valor diferenciador de otros enumerados y el valor como valor actual dentro de este. En memoria ambos pueden ser almacenados como números enteros ya que no es necesario saber su nombre original, tan solo su diferencia o similitud con otros de su tipo. Otros tipos como los **diccionarios** necesitarían de una implementación **más extensa y compleja** por su forma de operar y acceder a los datos.

5.3.3 Máquina Virtual

La máquina virtual implementa los **elementos básicos** y necesarios para funcionar, pero a su vez estos son suficientes para servir de base y **ampliar las utilidades** en un futuro. Los elementos con más potencial podrían considerarse: la entrada y salida virtual, el manejador de memoria y la CPU.

Virtual I/O

La entrada y salida virtual utiliza un segmento de la memoria para comunicarse, esto es útil para **datos pequeños** o para referencias al montículo, pero impide una conexión más fácil y fluida con **medios externos**. Al no poder acceder a la memoria del proceso es necesario realizar **un cambio en la implementación** por cada uno de los elementos que se requieran añadir a la máquina. Con una variación sobre este elemento, utilizando la escritura y lectura sobre un **archivo local** como medio, sería muy fácil recibir archivos de **grandes dimensiones** de manera **rápida**. Otra posible implementación sería utilizar la **conexión de un *socket*** para la transmisión de datos, aumentando mucho la **velocidad de transmisión** en serie de los mismos en ambos sentidos.

Memory Manager

El manejador de memoria utiliza distintas secciones de memoria para distintos fines: los registros, la pila y el montículo. Pero cada vez que un elemento es reservado en memoria ocupa al menos una posición, varias si este mismo elemento es utilizado a menudo en distintas funciones y muchas posiciones si el elemento es uno de uso habitual como por ejemplo el valor “True” o “False”. Para mejorar el uso de la memoria sería posible reservar una sección nueva, que contenga aquellos valores que aparezcan de forma habitual y no varíen. Esta sección de constantes podría inicializarse con los elementos habituales para la mayoría de programas y extenderse con aquellos que el desarrollador declare como tales. Por ejemplo: “True”, “False”, “0”, ‘1”, etc.

CPU

La CPU es la encargada del manejo de las instrucciones y su ejecución, pero su implementación actual solo permite un uso **secuencial** de sus recursos. En un futuro sería posible variar la manera en la que las instrucciones son procesadas con una **expresión interna** de la sintaxis del lenguaje que cree procesos nuevos capaces de ejecutar funciones en **paralelo**, aumentando la eficiencia del sistema en gran medida.

Apéndices

Apéndice A

Material adicional

A.1 Instrucciones

Byte-Code	Nombre	Argumentos	Descripción
00	NOP		Operación vacía
01	ADD	n m s	Reg[s] = Reg[n] + Reg[m]
02	ADDI	n m s	Reg[s] = n + Reg[m]
03	SUB	n m s	Reg[s] = Reg[n] - Reg[m]
04	SUBI	n m s	Reg[s] = n - Reg[m]
05	MULT	n m s	Reg[s] = Reg[n] * Reg[m]
06	MULTI	n m s	Reg[s] = n * Reg[m]
07	DIV	n m s	Reg[s] = Reg[n] / Reg[m]
08	DIVI	n m s	Reg[s] = n / Reg[m]
09	AND	n m s	Reg[s] = Reg[n] & Reg[m]
0a	ANDI	n m s	Reg[s] = n & Reg[m]
0b	OR	n m s	Reg[s] = Reg[n] | Reg[m]
0c	ORI	n m s	Reg[s] = n | Reg[m]
0d	EQ	n m s	Reg[s] = Reg[n] == Reg[m]
0e	EQI	n m s	Reg[s] = n == Reg[m]
0f	NEQ	n m s	Reg[s] = Reg[n] != Reg[m]
10	NEQI	n m s	Reg[s] = n != Reg[m]
11	LW	n m s	Reg[m] = Heap[Reg[n]]
12	LWI	n m s	Reg[m] = Heap[n]
13	SW	n m s	Heap[Reg[m]] = Reg[n]
14	SWI	n m s	Heap[m] = Reg[n]
15	PUSH	n	Push(Reg[n])
16	PUSHI	n	Push(n)
17	POP	n	Reg[n] = Pop()

Tabla A.2: Tabla de instrucciones completa (I).

18	JEQ	n m s	Si Reg[n] == Reg[m] entonces salta s instrucciones
19	JNE	n m s	Si Reg[n] != Reg[m] entonces salta s instrucciones
1a	J	n	Salta n instrucciones
30	JI	n	Salta a la instrucción n
1b	JR	n	IP = IP + Reg[n]
1c	JIZ	n m	Si Reg[n] == 0 entonces salta m instrucciones
1d	CAL	n	Guarda Stack Frame y salta a Reg[n]
1e	CALI	n	Guarda Stack Frame y salta a n
1f	RET	n	Guardar Reg[n] como ACC y cargar Stack Frame
20	ARG	n m	Reg[m] = Arg[n]
31	SARG	n m	Arg[n] = Reg[m]
21	MOV	n m	Reg[m] = n
22	MOVC	n m s	Const[m] = n
23	END	n m s	Fin
24	MAL	n m s	Reg[m] = Malloc(n)
25	MFRE	n m s	Free(Reg[n])
26	INP	n m s	Reg[n] = Inp(0)
	INPL	n m s	Input[0] n
28	OUT	n m s	Output[0] = Reg[n]
	OUTL	n m s	Output[0] = n
32	DBG	n m s	Debug()

Tabla A.4: Tabla de instrucciones completa (II).

Lista de acrónimos

ARM Advanced RISC Machines. 68

AST Abstract Syntax Tree. 68

BEAM Bogdan/Björn's Erlang Abstract Machine. 68

CISC Complex Instruction Set Computer. 68

JVM Java Virtual Machine. 68

MIPS Microprocessor without Interlocked Pipelined Stages. 68

OOP Object Oriented Programming. 68

PDF Portable Document Format. 68

POO Programación Orientada a Objetos. 68

RISC Reduced Instruction Set Computer. 68

VM Virtual Machine. 68

Glosario

azucar sintáctico Elementos añadidos a la sintaxis de un lenguaje de programación para hacer más sencillo su uso.. 68

Brew Administrador de paquetes para el sistema MacOs. 68

Byte-Code Código independiente de la máquina que generan los compiladores de determinados lenguajes (Java, Erlang,...) y que es ejecutado por el correspondiente intérprete. 3, 26, 68

C++ Lenguaje de programación basado en C. 68

C++ Lenguaje de programación basado en B. 5, 68

CPU Unidad central del procesamiento, encargada de procesar las instrucciones y acciones del sistema. 68

código abierto El código abierto es un modelo de desarrollo de software basado en la colaboración abierta para la que el contenido esta disponible al resto del mundo y pueden acceder a él.. 25, 68

Código Fuente Texto representativo de un programa escrito acorde a una gramática concreta y comprensible por el ser humano. 3, 68

Eof Marca final de un archivo. 68

expresión regular Conjunto de símbolos que se utilizan para detectar patrones. 4, 68

firma La firma de una función es el nombre argumentos y tipo a devolver de la misma.. 68

Frame Pointer Posición de memoria en la cual se almacena la posición de el Stack Pointer a la función actual. 68

Instruction Pointer Posición de memoria en la cual se almacena la posición de la instrucción actual a ejecutar. 68

Item Elemento de un conjunto. 68

JAVA Lenguaje de programación con un algoritmo orientado a objetos. 4, 5, 68

Lexer Programa dedicado al análisis léxico de un texto. 4, 68

licencia BSD Licencia empleada inicialmente por BSD que permite la distribución bajo unas condiciones establecidas en ella.. 25, 68

licencia GPL Licencia creada para el proyecto GNU que permite la distribución comercial.. 26, 68

MacOs Sistema operativo característico de los portátiles de la empresa Apple. 68

Memory Manager Elemento de una máquina virtual encargado del manejo de la memoria. 68

Parser Programa dedicado al análisis sintáctico de un texto. 4, 68

Path Dirección de recursos, referente a directorios, dentro de un ordenador. 68

pattern-matching Característica de algunos lenguajes funcionales por la cual se utiliza un patrón para realizar una comprobación. 1, 68

POSIX Familia de estándares especificadas por la IEEE con el objetivo de facilitar la interoperabilidad de sistemas operativos. 25, 68

Set Conjunto de elementos no repetidos. 68

snake case Convención de nombrado utilizando letras en minuscula y separaciones por medio de barras bajas.. 68

socket Estructura utilizada para intercambiar datos entre dos procesos.. 68

Stack Frame Sección de la pila de memoria dedicada a la representación de una función, el estado de los registros anterior y los datos de ejecución. 68

Stack Overflow Página web dedicada a la resolución de dudas en el campo de la informática y compartición de recursos. 68

Stack Pointer Posición de memoria en la cual se almacena la posición de la cabeza de la pila actual. 68

String Cadena de texto. 68

Tail Recursion Optimización de las llamadas recursivas de una función para minimizar el espacio de memoria utilizado. 68

tipado estático Caracteristica de un lenguaje de prgramación por la cual los tipos de sus elementos se conocen en el momento de su compilación.. 27, 68

Token Unidad mínima de un análisis sintáctico y elemento representativo de un elemento. 4, 5, 68

Bibliografía

[1] R. G. G. Cattell, *Formalization and automatic derivation of code generators*. UMI Research Press Ann Arbor, Michigan, 1982, no. 3.

[2] M. E. Lesk and E. Schmidt, *Lex: A lexical analyzer generator*. Bell Laboratories Murray Hill, NJ, 1975.

[3] J. Levine, *Flex & Bison: Text Processing Tools*. " O'Reilly Media, Inc.", 2009.

[4] N. Chomsky, "Three models for the description of language," *IRE Transactions on information theory*, vol. 2, no. 3, pp. 113–124, 1956.

[5] M. L. Minsky, *Computation*. Prentice-Hall Englewood Cliffs, 1967.

[6] D. J. Rosenkrantz and R. E. Stearns, "Properties of deterministic top down grammars," in *Proceedings of the first annual ACM symposium on Theory of computing*, 1969, pp. 165–180.

[7] S. JARZABEK and T. Krawczyk, "Ll-regular grammars," *Instytutu Maszyn Matematycznych*, p. 107, 1974.

[8] A. Aho, M. Lam, R. Sethi, and J. Ullman, "Compilers: Principles, techniques and tools, 2nd editio," 2007.

[9] N. P. Chapman, *LR parsing: theory and practice*. CUP Archive, 1987.

[10] S. Khuri and J. Williams, "Understanding the bottom-up slr parser," *ACM SIGCSE Bulletin*, vol. 26, no. 1, pp. 339–343, 1994.

[11] F. DeRemer and T. Pennello, "Efficient computation of lalr (1) look-ahead sets," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 4, no. 4, pp. 615–649, 1982.

[12] M. Tomita, *Generalized LR parsing*. Springer Science & Business Media, 1991.

[13] J. R. Levine, J. Mason, J. R. Levine, T. Mason, D. Brown, and P. Levine, *Lex & yacc.* " O'Reilly Media, Inc.", 1992.

[14] S. C. Johnson *et al.*, *Yacc: Yet another compiler-compiler.* Bell Laboratories Murray Hill, NJ, 1975, vol. 32.

[15] R. P. Corbett, "Static semantics and compiler error recovery," CALIFORNIA UNIV BERKELEY DEPT OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCES, Tech. Rep., 1985.

[16] T. J. Parr and R. W. Quong, "Antlr: A predicated-ll (k) parser generator," *Software: Practice and Experience*, vol. 25, no. 7, pp. 789–810, 1995.

[17] T. Parr, S. Harwell, and K. Fisher, "Adaptive ll (*) parsing: the power of dynamic analysis," *ACM SIGPLAN Notices*, vol. 49, no. 10, pp. 579–598, 2014.

[18] J. Jones, "Abstract syntax tree implementation idioms," in *Proceedings of the 10th conference on pattern languages of programs (plop2003)*, 2003, pp. 1–10.

[19] T. Æ. Mogensen, "Machine-code generation," in *Introduction to Compiler Design.* Springer, 2011, pp. 147–157.

[20] R. Wilhelm and H. Seidl, *Compiler design: virtual machines.* Springer Science & Business Media, 2010.

[21] T. Peters, "The zen of python," in *Pro Python.* Springer, 2010, pp. 301–302.

[22] P. S. Foundation. (2015) What's new in python 3.5. [En línea]. Disponible en: https://docs.python.org/3.5/whatsnew/3.5.html

[23] D. Thomas, *Programming Elixir≥ 1.6: Functional|> Concurrent|> Pragmatic|> Fun.* Pragmatic Bookshelf, 2018.

[24] S. Muchnick *et al.*, *Advanced compiler design implementation.* Morgan kaufmann, 1997.

[25] T. Lindholm, F. Yellin, G. Bracha, and A. Buckley, *The Java Virtual Machine Specification, Java SE 7 Edition: Java Virt Mach Spec Java_3.* Addison-Wesley, 2013.

[26] S. Overflow. (2022) 2022 developer survey. [En línea]. Disponible en: https://survey.stackoverflow.co/2022/#most-popular-technologies-language