INSTITUTO TECNOLÓGICO DE CELAYA

Ingeniería en Sistemas Computacionales

Lenguajes y Autómatas II

Proyecto Final MCScript

Equipo 5

Ulises Andrade González
Emilio Sebastián Chávez Vega
Luis Ángel Quijano Guerrero
Jesús Adrian Pacheco Garcia
Emiliano Rebolledo Navarrete

Fecha: Julio 2025

3. Introducción

En el ámbito de la computación, los compiladores desempeñan un papel fundamental al traducir lenguajes de alto nivel a un lenguaje comprensible por la máquina. Este proceso es esencial en el desarrollo de software, ya que, de no existir los compiladores, las instrucciones tendrían que escribirse directamente en código binario. Incluso el lenguaje ensamblador, que es el nivel más bajo de programación humana, necesita ser traducido y compilado para ser ejecutado por la máquina.

Este documento presenta la documentación del desarrollo de un compilador diseñado como parte del proyecto final de la asignatura Lenguajes y Autómatas II. El compilador interpreta un lenguaje propio, inspirado en Java y Python, que combina características como el tipado débil de Python y la estructura de bloques de Java. Este lenguaje fue pensado con un enfoque lúdico y educativo, orientado a niños y jugadores del popular videojuego Minecraft.

A lo largo del documento se presenta el marco teórico que sustenta el diseño e implementación del compilador, la definición formal del lenguaje desarrollado, así como los objetivos, alcances y limitaciones del proyecto. Además, se incluye la implementación completa del compilador, con su respectiva documentación, que detalla cada una de las fases del proceso de compilación.

4. Marco teórico

4.1 Fundamentos de los Compiladores

4.1.1 Definición

Un compilador es un programa informático cuya función central es traducir un programa escrito en un lenguaje fuente a un lenguaje objeto, generalmente de un lenguaje de alto nivel comprensible para humanos a uno de bajo nivel como código máquina o ensamblador, que puede ser ejecutado directamente por una computadora. Este proceso no sólo implica la traducción directa de instrucciones, sino también la verificación de la corrección sintáctica y semántica, así como la optimización del código resultante. El uso de compiladores ha sido fundamental en el desarrollo de la informática, ya que permite que los programadores utilicen lenguajes más expresivos y abstractos, facilitando así el desarrollo de software complejo y portable a diferentes arquitecturas (Aho, Lam, Sethi & Ullman, 2007).

La importancia de los compiladores radica en que actúan como un puente entre el código abstracto hecho por una persona y la ejecución concreta por parte de una máquina. Esto ha permitido la evolución de los lenguajes de programación, propiciando la creación de lenguajes especializados para distintos dominios de aplicación, lo cual sería inviable si cada lenguaje

careciera de una forma sistemática de traducción a instrucciones comprensibles para la computadora (Aho et al., 2007).

4.1.2 Fases de la Compilación

El proceso de compilación se compone de una serie de fases o etapas claramente definidas. Cada una de ellas cumple una función específica El entendimiento profundo de estas fases es esencial para el diseño e implementación eficiente de un compilador (Aho, Lam, Sethi & Ullman, 2007; Grune, Bal, Jacobs & Langendoen, 2012).

1. Análisis Léxico

El análisis léxico es la primera fase del proceso de compilación. Su función es leer el código fuente como una secuencia de caracteres de izquierda a derecha y agruparlos en unidades significativas llamadas tokens, que representan palabras clave, identificadores, operadores, signos de puntuación, llaves y otros símbolos del lenguaje. Para llevar a cabo esta tarea, se pueden emplear expresiones regulares y autómatas finitos, permitiendo al compilador reconocer patrones y estructuras básicas (Aho et al., 2007).

El analizador léxico también se encarga de eliminar comentarios y espacios en blanco, así como de reportar errores relacionados con símbolos no reconocidos o mal formados. Además, suele construir una tabla de símbolos inicial que registra información relevante sobre los identificadores encontrados, aunque no es completamente necesario. Un analizador léxico bien diseñado mejora la eficiencia de las etapas posteriores al simplificar la estructura del código fuente (Grune et al., 2012).

2. Análisis Sintáctico

El análisis sintáctico es la segunda fase del compilador y tiene como propósito determinar si la secuencia de tokens producida por el análisis léxico cumple con las reglas gramaticales del lenguaje de programación. Esta fase construye una representación jerárquica conocida como árbol de sintaxis abstracta (AST) (Aho et al., 2007).

Para ello, se emplean gramáticas libres de contexto y algoritmos de análisis sintáctico, como los analizadores descendentes que construye el AST del nodo inicial para abajo (LL) y ascendentes (LR). El analizador sintáctico también identifica errores estructurales, tales como la falta de paréntesis o el orden incorrecto de sentencias, proporcionando mensajes útiles para su corrección y se suele detener la fase de compilación aqui si se detectan errores léxico o sintácticos (Grune et al., 2012).

El AST generado es fundamental para las etapas posteriores, ya que contiene toda la información necesaria sobre la organización y la relación entre las distintas partes del programa.

3. Análisis Semántico

En la fase de análisis semántico se valida que las construcciones sintácticamente correctas tengan sentido o significado correcto dentro del contexto del lenguaje y del programa específico. Esto incluye la verificación de tipos de datos, comprobación de compatibilidad en operaciones, existencia y visibilidad de variables y funciones, así como el cumplimiento de restricciones semánticas definidas en la especificación del lenguaje (Aho et al., 2007).

Durante esta fase, donde se recorre el AST y que inicia en el nodo inicial del AST, el compilador suele utilizar una tabla de símbolos, que almacena información sobre los identificadores del programa (nombres, tipos, ámbitos, etc.). El análisis semántico previene errores lógicos que no pueden ser detectados por el análisis sintáctico, como el uso de variables no declaradas, la incompatibilidad de tipos en asignaciones, o el número incorrecto de argumentos en una llamada a función (Grune et al., 2012). Este análisis puede proporcionar información adicional al AST para las fases de síntesis, e incluso se puede descomponer esta fase adicionando las tareas semánticas en el análisis sintáctico y en la generación de código intermedio, aunque claro puede crearse como etapa completamente separada, en cuyo caso, si se detectan errores, no se pasa a las siguientes fases.

4. Generación de Código Intermedio

Tras completar el análisis semántico, el compilador traduce el programa a una representación intermedia, que es un nivel de abstracción entre el código fuente y el código máquina. Esta representación suele ser independiente de la arquitectura de la computadora objetivo y facilita tanto la portabilidad del compilador como la aplicación de optimizaciones (Aho et al., 2007). El código intermedio es más abstracto que el fuente pero más concreto que el objeto, si es que el objeto es código máquina.

Las formas comunes de código intermedio incluyen el código de tres o cuatro direcciones, árboles de sintaxis anotados, o representaciones tipo bytecode. El código intermedio preserva la estructura lógica y semántica del programa original, pero es más fácil de manipular y transformar para las fases siguientes, además de que se pueden hacer tareas especificas a la estructura que se desea y funcionalidad del lenguaje en forma de instrucciones adicionales a tareas, como casteos. (Muchnick, 1997).

5. Optimización de Código

La optimización de código es una etapa dedicada a mejorar la eficiencia del programa intermedio antes de generar el código máquina final. El objetivo es reducir el tiempo de ejecución, el uso de memoria o ambos, sin modificar el comportamiento observable del programa (Aho et al., 2007).

Existen dos tipos principales de optimización: optimización local (dentro de un bloque de código) y optimización global (a lo largo de todo el programa). Entre las técnicas comunes se encuentran la eliminación de código muerto, la propagación de constantes, la optimización de

bucles, y la minimización de accesos a memoria. La optimización es crucial en aplicaciones que requieren alto rendimiento y eficiencia de recursos (Muchnick, 1997).

6. Generación de Código Objeto

En la fase de generación de código objeto o código máquina, el compilador traduce el código intermedio optimizado a instrucciones específicas para la arquitectura hardware destino (CPU, microcontrolador, etc.). Aquí se asignan registros, se gestionan las direcciones de memoria y se construyen los archivos ejecutables o ensamblador final (Aho et al., 2007). También y como simulación, puede que el lenguaje destino no sea de bajo nivel si no otro lenguaje, en este caso se utilizan traducciones de cada instrucción intermedia además del uso de estructuras de datos simulando pilas y registros si fuesen necesarias.

La calidad del código objeto generado impacta directamente en el rendimiento del programa, por lo que se aplican técnicas adicionales como la asignación eficiente de registros, la selección de instrucciones y el manejo adecuado de saltos y llamadas a funciones. El resultado de esta fase es un archivo ejecutable que puede ser corrido directamente en la máquina objetivo (Grune et al., 2012).

4.1.3 Conceptos Clave

La construcción de compiladores es una disciplina que integra diversos conceptos teóricos y prácticos de las ciencias de la computación. Entre los términos fundamentales se encuentran los autómatas, las gramáticas, las estructuras de representación de programas y las estructuras de datos que permiten la gestión eficiente de la información durante la compilación. A continuación se describen los conceptos clave más relevantes.

1. Autómata Finito Determinista (AFD)

Un Autómata Finito Determinista (AFD) es un modelo matemático que se utiliza para reconocer lenguajes regulares, es decir, conjuntos de cadenas que pueden ser descritas mediante expresiones regulares. Un AFD consiste en un conjunto finito de estados, un alfabeto de entrada, una función de transición, un estado inicial y un conjunto de estados de aceptación. En el contexto de compiladores, los AFDs son esenciales en la fase de análisis léxico, ya que permiten identificar y clasificar los distintos tokens en el texto fuente (Hopcroft, Motwani & Ullman, 2007).

El uso de AFD permite construir analizadores léxicos eficientes y automáticos, capaces de escanear grandes volúmenes de código en tiempo lineal respecto a la longitud de la entrada (Aho, Lam, Sethi & Ullman, 2007).

2. Gramática Libre de Contexto (GLC)

Una Gramática Libre de Contexto (GLC) es un sistema formal compuesto por un conjunto de reglas de producción que definen cómo se pueden generar cadenas válidas en un lenguaje. Las GLC son la base para describir la sintaxis de los lenguajes de programación y son

especialmente adecuadas para representar estructuras anidadas y jerárquicas, como expresiones aritméticas, sentencias condicionales o bloques de código (Hopcroft et al., 2007).

Las GLC son fundamentales en la construcción de analizadores sintácticos, pues permiten automatizar el reconocimiento de la estructura gramatical del código fuente y la generación de árboles de derivación y sintaxis abstracta (Aho et al., 2007).

3. Árbol de Sintaxis Abstracta (AST)

El Árbol de Sintaxis Abstracta (AST, por sus siglas en inglés) es una estructura jerárquica que representa la organización lógica de un programa fuente. A diferencia del árbol de derivación o sintaxis completa, el AST elimina detalles sintácticos innecesarios (como paréntesis o ciertos delimitadores) y se centra en las relaciones semánticas y operativas entre los componentes del código (Grune, Bal, Jacobs & Langendoen, 2012).

El AST es utilizado por las fases posteriores de la compilación, como el análisis semántico, la generación de código intermedio y las optimizaciones, ya que facilita el recorrido y la manipulación estructurada del programa (Aho et al., 2007).

4. Tabla de Símbolos

La tabla de símbolos es una estructura de datos utilizada por el compilador para almacenar información sobre los identificadores encontrados en el código fuente, tales como variables, constantes, funciones, clases, etc. Para cada identificador, la tabla puede registrar atributos como el tipo de dato, ámbito, dirección de memoria, valor actual, entre otros (Aho et al., 2007).

La tabla de símbolos es esencial para el análisis semántico, la verificación de declaraciones, scope y usos, la comprobación de tipos y la generación de código. Una implementación eficiente de esta estructura permite mejorar significativamente el rendimiento y la calidad del compilador (Grune et al., 2012).

5. Ámbito (Scope)

El ámbito o scope se refiere a la región del programa donde un identificador (variable, función, etc.) es visible y puede ser utilizado. Los compiladores gestionan el ámbito para prevenir conflictos de nombres y garantizar el uso correcto de las declaraciones en el contexto adecuado (Scott, 2009).

6. Tipos de Datos y Verificación de Tipos

La verificación de tipos es el proceso mediante el cual el compilador asegura que las operaciones se realizan entre operandos de tipos compatibles. Los sistemas de tipos son una parte central de los lenguajes modernos y su correcta gestión es fundamental para prevenir errores y vulnerabilidades (Scott, 2009).

7. Errores Léxicos, Sintácticos y Semánticos

Durante la compilación, pueden detectarse errores léxicos (secuencias no reconocidas como tokens válidos), errores sintácticos (estructuras que violan las reglas gramaticales) y errores semánticos (incongruencias en el significado o uso de los elementos del programa). Un buen compilador proporciona información clara y detallada para la corrección de estos errores (Aho et al., 2007). También se pueden presentar errores durante la ejecución que deben ser previstos para el correcto comportamiento del compilador.

4.2 Diseño de Lenguajes y Temática

El diseño de lenguajes de programación es una rama fundamental de la informática que implica la definición formal de la sintaxis y semántica de un lenguaje, la especificación de su funcionalidad, y la forma en la que los usuarios interactuarán con él. Al diseñar un lenguaje, se deben considerar aspectos como la facilidad de aprendizaje, la expresividad, la seguridad, el modo y eficiencia de ejecución y las necesidades del público objetivo (Scott, 2009).

Uno de los retos más relevantes es lograr que el lenguaje sea adecuado para el tipo de tareas o problemas que busca resolver. Para ello, se definen estructuras de control, tipos de datos, convenciones de nomenclatura y mecanismos de interacción que respondan a un propósito concreto. Además, la inspiración en temáticas populares o contextos específicos, como Minecraft, puede favorecer el interés y la motivación de los usuarios, en especial en entornos educativos (Fowler, 2010).

4.2.1 Lenguajes de Dominio Específico (DSL)

Un Lenguaje de Dominio Específico (Domain-Specific Language, DSL) es un lenguaje de programación o especificación diseñado para abordar problemas concretos dentro de un dominio particular. A diferencia de los lenguajes de propósito general (como Python, Java o C), los DSL ofrecen una sintaxis y semántica especializadas que simplifican tareas recurrentes en ese dominio, mejorando la productividad y reduciendo la posibilidad de errores (Mernik, Heering & Sloane, 2005).

Los DSL pueden ser externos (con su propia sintaxis y gramática). Su principal ventaja radica en que permiten a los usuarios expresar soluciones de forma más natural y directa, alineada con los conceptos propios del dominio de aplicación (Van Deursen, Klint & Visser, 2000).

En contextos educativos, el uso de DSL facilita que los aprendices se enfoquen en los conceptos clave del dominio generando más interés y sin verse abrumados por la complejidad de lenguajes tradicionales, promoviendo la experimentación, la creatividad y el aprendizaje activo (Fowler, 2010).

4.2.2 Minecraft

Minecraft es un videojuego de construcción y aventura lanzado por Mojang en 2011, que permite a los jugadores interactuar con un mundo tridimensional compuesto por bloques. Los usuarios pueden recolectar recursos, construir estructuras, explorar escenarios, enfrentarse a criaturas y colaborar con otros jugadores. Minecraft se ha destacado no solo como juego de entretenimiento, sino también como una plataforma educativa, debido a su flexibilidad, posibilidades de modificación (mods) y apoyo a la creatividad y el pensamiento computacional (Short, 2012).

El entorno de Minecraft permite modificar o automatizar acciones mediante scripts o lenguajes de programación específicos, ya sea a través de su API, mods o servidores personalizados. Esto lo ha convertido en un espacio propicio para la enseñanza de programación y conceptos de ciencias de la computación, ya que los estudiantes pueden ver de forma inmediata y visual el resultado de su código (Clark & Bower, 2021).

4.3 Tecnologías y Herramientas Utilizadas

4.3.1 Python como Lenguaje de Implementación

Python es actualmente uno de los lenguajes de programación más populares y versátiles para el desarrollo de sistemas de software, incluidos compiladores y herramientas de procesamiento de lenguajes. Su sintaxis sencilla, amplia comunidad y extenso ecosistema de bibliotecas lo hacen ideal tanto para la prototipación rápida como para la implementación de proyectos complejos (Van Rossum & Drake, 2009).

El diseño flexible de Python permite manipular fácilmente cadenas, estructuras de datos complejas y algoritmos, características fundamentales para la construcción de compiladores y analizadores de lenguajes. Python también se destaca por su integración con otras tecnologías y su capacidad para servir como “pegamento” entre diferentes componentes de software, lo cual resulta muy útil en el desarrollo de aplicaciones web modernas que combinan lógica de negocio, procesamiento de lenguaje y presentación gráfica (Lutz, 2021).

4.3.2 Bibliotecas y Módulos Específicos

El desarrollo de un compilador o sistema de procesamiento de lenguajes moderno frecuentemente requiere exponer su funcionalidad a través de una interfaz web, lo que implica la combinación de varias tecnologías y módulos. En este contexto, Python funge como el núcleo lógico y de procesamiento, mientras que otras tecnologías complementan la experiencia del usuario.

1. FastAPI

FastAPI es un framework web moderno, rápido y eficiente para Python, que permite crear APIs web robustas de manera sencilla. Gracias a su rendimiento, basado en ASGI y en tipado estático, FastAPI es ideal para exponer funcionalidades de compiladores o sistemas expertos a través de servicios REST o WebSockets (Ramírez, 2019). Facilita la integración de lógica Python en la web y la comunicación entre el frontend y el backend.

2. Jinja2

Jinja2 es un motor de plantillas ampliamente utilizado para conectar Python con HTML. Permite generar contenido HTML dinámico a partir de variables y estructuras de datos de Python, facilitando la creación de interfaces web interactivas y personalizadas sin perder la separación lógica entre presentación y procesamiento (Ronacher, 2008).

3. HTML, CSS y JavaScript

El HTML es el lenguaje de marcado estándar para la estructura de páginas web, CSS se utiliza para la presentación visual y el diseño, y JavaScript agrega interactividad y dinamismo en el frontend. Aunque estas tecnologías no pertenecen a Python, son imprescindibles para construir aplicaciones web usables y atractivas, permitiendo la comunicación en tiempo real con el backend Python (Flanagan, 2020; Duckett, 2011).

En el contexto del desarrollo de un compilador en entorno web, JavaScript puede manejar eventos de usuario, enviar solicitudes al backend implementado en Python/FastAPI, y actualizar la interfaz en tiempo real según la respuesta del compilador.

4. Integración de Todas las Tecnologías

En el flujo de trabajo típico, Python gestiona el procesamiento principal (análisis, compilación, validación), FastAPI expone esta funcionalidad a través de una API o aplicación web, Jinja2 se encarga de la generación de vistas y páginas HTML basadas en los resultados de Python, y JavaScript asegura la interacción dinámica y la actualización instantánea de la interfaz para el usuario final.

Este enfoque modular y tecnológico nos permitió desarrollar nuestro compilador de tal manera que sea accesible, interactivo y con alto impacto educativo y práctico, aprovechando lo mejor de cada tecnología.

5. Desarrollo del caso de estudio

5.1 Definición del Lenguaje

Nombre del compilador: MCScript

Logo: Por medio del logo buscamos enfatizar en la inspiración que tuvimos del juego Minecraft para el desarrollo de este proyecto, utilizando elementos del juego que de igual forma se utilizan dentro de nuestro compilador.

Imagen 1. Logo de MCScript

Slogan: “Construye tu código, bloque a bloque.”

Descripción del lenguaje y compilador:
MCScript es un lenguaje de programación y compilador educativo e introductorio inspirado en Minecraft, diseñado para enseñar estructuras de programación mientras los usuarios sienten que están "crafteando" código usando conceptos del juego.

Combina conceptos fundamentales de Minecraft con la compilación compilación como:

- Bloque = enteros
- Losa = real
- Palanca = booleano
- Cofres = listas
- Item = tipo débil
- Portal = funciones

Características clave:

- Sintaxis sencilla basada en Python y Java con adiciones que suelen estar presentes en muchos otros lenguajes, como uso de corchetes, punto y coma, tipado fuerte.
- Relacionar conceptos clave de programación con conceptos del juego, como concatenación de strings como concatenación de hojas en un libro.
- Soporte para condiciones, ciclos, funciones y estructuras de datos como listas.
- Interfaz de uso llamativa, con demostración de cada fase del compilador.
- Es una traducción de MCScript a un código ejecutable de Python.

5.2 Objetivos

General

Desarrollar un lenguaje de programación inspirado en el videojuego de Minecraft con el fin de introducir a los niños y principiantes en programación, abstrayendo conceptos de programación con equivalentes en Minecraft para que el aprendizaje y transición hacia otros lenguajes sea más amena y sencilla.

Especificos

- Definir los conceptos de Minecraft que se usarán para la programación.
- Definir las palabras reservadas, la sintaxis y la gramática que usará el lenguaje.
- Crear un interfaz sencilla para que el usuario pueda escribir su código y ver sus resultados y las fases del compilador en las diferentes pestañas de la consola.
- Ejecutar las cuatro operaciones aritméticas básicas, ya sea con nuestras funciones integradas o por expresiones.

5.3 Alcances

Tecnológico

MCScript, mediante la implementación de los siguientes conceptos fundamentales, establece las bases de un lenguaje de programación completo, pero accesible. Estos alcances permiten que el usuario pueda desarrollar lógica básica en un entorno amigable.

- Decisiones básicas
- Declaración de función básica
- Estructura de datos básicas sin métodos
- Operaciones básicas

Social

MCScript tiene el potencial de impactar socialmente al unir dos conceptos claves en la actualidad, la programación y los conceptos de un videojuego tan popular como Minecraft. Al ofrecer un lenguaje de programación simple, accesible, educativo y gamificado, este proyecto puede abrir puertas al conocimiento tecnológico a comunidades que en un inicio no podrían verse tan interesados en el aprendizaje de un lenguaje de programación.

Limitaciones

El lenguaje se limita a poder crear código secuencial, ya que no posee soporte para paradigmas de programación como orientación a objetos, funcional o lógico; se limita a tener tipado de datos básicos como entero, flotante, char, string y un tipado débil llamado item.
Posee las estructuras de control básicas como condiciones “si”, “si no”, bucles básicos como

“para” y “mientras”. Además, únicamente tiene una estructura de datos lista a la que se le llama “cofre”.

5.4 Viabilidad del lenguaje

La viabilidad de MCScript se debe analizar desde dos perspectivas: como herramienta educativa y como lenguaje de propósito general.

Viabilidad como Herramienta Educativa
En su rol educativo, MCScript es altamente viable. Su diseño cumple con los objetivos pedagógicos para la enseñanza de los fundamentos de la programación.

- Sintaxis Intuitiva y Temática: El uso de palabras clave inspiradas en Minecraft (craftear, romper, cofre, portal) hace que el lenguaje sea más atractivo y menos intimidante para los estudiantes que se inician en la programación. Esto facilita la comprensión de conceptos abstractos al asociarlos con elementos familiares.

- Enfoque en la Programación Estructurada: El lenguaje se centra en enseñar los pilares de la programación: variables, tipos de datos, estructuras de control (si, mientras, para) y funciones. Al limitar las características avanzadas, permite que los estudiantes se concentren en dominar estos conceptos fundamentales.

- Curva de Aprendizaje Suave: La simplicidad del lenguaje y la exclusión de paradigmas complejos (como la orientación a objetos) aseguran una curva de aprendizaje gradual. Esto es ideal para cursos introductorios donde el objetivo principal es desarrollar el pensamiento algorítmico.

Viabilidad como Lenguaje de Propósito General
Como lenguaje para desarrollar aplicaciones del mundo real, la viabilidad de MCScript es limitada. Su diseño educativo introduce restricciones que lo hacen poco práctico para proyectos complejos.

- Falta de Biblioteca Estándar: La ausencia de un conjunto de bibliotecas para tareas comunes (manejo de archivos, redes, interfaces gráficas) significa que cualquier funcionalidad más allá de la lógica básica debe ser implementada desde cero, lo cual es ineficiente.

- Gestión de Errores Primitiva: MCScript carece de mecanismos para la gestión de errores en tiempo de ejecución, como bloques try-catch. Esto hace que la creación de software robusto y fiable sea un desafío.

- Rendimiento y Escalabilidad: Al ser un lenguaje interpretado y sin optimizaciones avanzadas, su rendimiento no es comparable al de lenguajes compilados o de sistemas.

No sería adecuado para aplicaciones que requieren alta velocidad o manejo de grandes volúmenes de datos.

- Ecosistema Inexistente: Un lenguaje de propósito general depende de su comunidad, herramientas (depuradores, linters, gestores de paquetes) y documentación. MCScript, al ser un proyecto académico, no cuenta con este ecosistema.

5.5 Perfil de usuario

El usuario ideal de MCScript es una persona:

- Creativa
- Curiosa
- Que reconozca los términos del videojuego Minecraft
- Que tenga el interés emergente por la programación.
- Un estudiante o educador que desee aprender los fundamentos de la programación o desee enseñarla de una forma fácil y asociativa a conceptos de Minecraft.

Es importante que esta persona cuente con una computadora para poder utilizar el programa y que sepa escribir.

5.6 Elementos del lenguaje

5.6.1 Palabras reservadas (inicio, fin, si entonces..., etc.)

- SPAWNEAR: Es la palabra con la que debe de iniciar el programa, como spawnear en el mundo de minecraft.
- MORIR: Es la palabra con la que debe de terminar el programa, como morir en el mundo de Minecraft.
- CRAFTEAR: Es la palabra reservada para la función que suma dos parámetros.
- ROMPER: Es la palabra reservada para la función que resta dos parámetros.
- APILAR: Es la palabra reservada para la función que multiplica dos parámetros.
- REPARTIR: Es la palabra reservada para la función que divide dos parámetros.
- SOBRAR: Es la palabra reservada para la función que divide dos parámetros y devuelve el sobrante.
- ENCANTAR: Es la palabra reservada para la función que eleva a la potencia que indique el segundo parámetro, el primer parámetro.
- ANTORCHAR: Es la palabra reservada para la negación (NOT LÓGICO)
- PORTAL: Es la palabra reservada para definir una nueva función.
- TELETRANSPORTAR: Es la palabra reservada para retornar un valor de una función.
- SI: Es la palabra reservada para la sentencia condicional (IF).
- SINO: Es la palabra reservada para la sentencia condicional (ELSE).
- MIENTRAS: Es la palabra reservada para el ciclo WHILE.
- PARA:Es la palabra reservada para el ciclo FOR.
- CHAT: Es la palabra reservada para imprimir algo en la consola.

- CARTEL: Es la palabra reservada para leer algo de la consola.
- BLOQUE: Es la palabra reservada para el tipo de dato entero.
- LOSA: Es la palabra reservada para el tipo de dato real.
- HOJA: Es la palabra reservada para el tipo de dato de carácter.
- LIBRO: Es la palabra reservada para el tipo de dato cadena de texto.
- COFRE: Es la palabra reservada para las listas.
- ITEM: Es la palabra reservada para el tipado débil.
- PALANCA: Es la palabra reservada para el tipo de dato booleano.
- ENCENDIDO: Es la palabra reservada para referirse al TRUE.
- APAGADO: Es la palabra reservada para referirse al FALSE.

5.6.2 Identificadores

Para los identificadores se siguen las siguientes reglas léxicas:

- Solo acepta letras, números y guión bajo.
- No se puede iniciar con un numéro.

Ejemplo:

- Valido: Madera1, algo_3
- No valido: 1x, d-d

5.6.3 Caracteres especiales

- “”: Cualquier cosa que se ponga entre comillas dobles será identificada como un String dentro del lenguaje.
- //: La doble diagonal se usa para hacer comentarios dentro del código, cualquier cosa escrita con precedencia de estas diagonales será ignorada completamente.
- ,: Se utiliza para separar los datos cuando es que se necesita dentro de la sintaxis, por ejemplo al utilizar craftear(a,b), se utiliza para separar los valores.
- .: Se utiliza para concatenar elementos.
- {: Cuando se inicia una condición, un ciclo o un inicio de código, se utiliza para identificar dónde es que inicia lo que se va a realizar esta parte del código.
- }: Cuando se termina una condición, un ciclo o un inicio de código, se utiliza para identificar dónde es que termina lo que se va a realizar esta parte del código.
- (: Cuando se necesita enviar un valor a una función o en una declaración se utiliza este para delimitar el inicio de los parámetros que se van a enviar, dentro de este se introducen los valores y se separan por “,”.
-): Cuando se necesita enviar un valor a una función o en una declaración, se utiliza este para delimitar el final de los parámetros que se van a enviar.
- ;: Cuando se quiere terminar con un fragmento de código se utiliza este para separar las líneas.

5.6.4 Constantes.

No existe soporte para constantes de cualquier tipo dentro del lenguaje.

5.6.5 Variables.

Se usan tipos con tipado fuerte como los siguientes:

- Bloque: esta palabra clave representa el tipo de dato entero dentro del lenguaje.
- Losa: esta palabra se usa para definir cualquier número real (flotante).
- Hoja: este tipo es el equivalente al tipo char de otros lenguajes, solo acepta un carácter y debe ir entre comillas dobles
- Libro: este tipo es el equivalente al tipo String y se le puede asignar cualquier cadena de caracteres (deben ir entre comillas dobles).
- Palanca: este tipo es el equivalente al booleano, únicamente acepta valores verdadero o falso (encendido o apagado respectivamente).
- Cofre: este como tal no es un tipo de dato, sino que actúa como una lista a la cual puede asignársele cualquier otro tipo de dato.

También existe un tipo con tipado débil con la intención de hacer más fácil el desarrollo si fuese requerido:

- Item: puede ser cualquier tipo menos función.

5.6.6 Expresiones.

Las expresiones siguen la siguiente gramática:

expresion	→	expresion_logica
expresion_logica	→	expresión_igualdad resto_lógica
resto_logica	→	(Y | O) expresion_igualdad resto_logica | ε
expresion_igualdad	→	expresion_relacional resto_igualdad
resto_igualdad	→	== expresion_relacional resto_igualdad | ε
expresion_relacional	→	expresion_aritmetica resto_relacional
resto_relacional	→	(< | > | <= | >=) expresion_aritmetica resto_relacional | ε

expresion_aritmetica → termino resto_aritmetico

---

resto_aritmetico → (+ | -) termino resto_aritmetico
| ε

---

termino → exponente resto_termino

---

resto_termino → (* | / | %) exponente resto_termino
| ε

---

exponente → factor resto_exponente

---

resto_exponente → ^ factor resto_exponente
| ε

---

factor → (expresion)
| ID
| NUM
| (-|+) NUM
| ID [expresion]
| ENCENDIDO
| APAGADO
| llamada_funcion
| cadena_texto

Y a una sentencia de asignación o de declaración le sigue una expresión que se puede derivar en la anterior gramática.

5.6.7 Instrucciones.

El lenguaje tiene las palabras reservadas de SPAWNEAR y MORIR para indicar el inicio y fin de código como se muestra en la siguiente gramática:

programa → SPAWNEAR { lista_sentencias } MORIR;

El lenguaje tiene soporte para estructuras condicionales y bucles siguiendo la siguientes sintaxis:

sentencia_si → SI (expresion) { lista_sentencias } SINO { lista_sentencias }
| SI (expresion) { lista_sentencias }
| SI (expresion) { lista_sentencias } SINO sentencia si

---

sentencia_mientras → MIENTRAS (expresion) { lista_sentencias }

---

sentencia_para → PARA (sentencia_asignacion ; expresion ; sentencia_asignacion) { lista_sentencias }

5.6.8 Operadores

- =: Se utiliza para asignar un valor a una variable.
- ==: Se utiliza para para comparar los valores dentro del código y que estos sean iguales, esto nos retorna una expresión booleana verdadera en caso de que coincidan y falsa en caso de que no coincidan.
- <: Se utiliza para comparar que el primer valor sea menor al segundo, este nos retornará una expresión booleana verdadera en caso de serlo y falsa en caso de que sea mayor o igual.
- <=: Se utiliza para comparar que el primer valor sea menor o igual al segundo, este nos retornará una expresión booleana verdadera en caso de serlo y falsa en caso de que sea mayor.
- >: Se utiliza para comparar que el primer valor sea mayor al segundo, este nos retornará una expresión booleana verdadera en caso de serlo y falsa en caso de que sea menor o igual.
- >=: Se utiliza para comparar que el primer valor sea mayor al segundo, este nos retornará una expresión booleana verdadera en caso de serlo y falsa en caso de que sea menor.
- ^: Se utiliza para elevar el valor de el primer operador a la potencia del segundo operador, ambos operadores pueden tomar el valor de variables numéricas.
- %: Al igual que python se utiliza para formatear cadenas cuando se combina con una s después del símbolo “%s”, cuando se combina con una i se utiliza para indicar el formato de números enteros “%i” y por último se utiliza con una d para enfocar los enteros decimales “%d”
- *: Es una de las alternativas que tenemos dentro del compilador para multiplicar un valor con otro.
- +: Es una de las alternativas que tenemos dentro del compilador para sumar un valor con otro.

- -: Es una de las alternativas que tenemos dentro del compilador para restar un valor con otro.

5.6.9 Consideraciones semánticas

- Antes de asignar un valor a una variable ésta debe de estar declarada.
- Debe haber compatibilidad de tipos de datos entre el que se declara y el que se asigna. A excepción del tipo de dato ITEM.
- No se puede repetir la declaración de un identificador en el mismo scope, pero si en diferentes.
- La expresión que se evalúa en las condiciones de los ciclos y sentencia SI debe de ser tipo palanca o ítem.

6. Código documentado

6.1 Analizador léxico

El programa inicia leyendo la matriz de transición de nuestro autómata finito para clasificar y detectar los tokens en el código fuente. *(Link autómata: https://lucid.app/lucidchart/b7a5357e-cd19-433b-9d3d-6322f9b0e550/edit?invitationId=inv_30707eb9-f64c-4d8a-8100-7f13bb16207b&page=0_0#)*
Genera una clase automata con atributos como definición formal de autómata, con estados, matriz de transición, estado inicial, estados finales, alfabeto. Todos estos elementos, junto con el código fuente, lo usa con el siguiente algoritmo. Este consiste en simplemente seguir los estados del autómata según el estado en el que está y lo que llega, los ifs que contiene son para casos especiales de nuestro lenguaje.

```
def run(self, text):
    list = []
    errors = []
    i = 0
    column = 1
    row = 1
    while i < len(text):
        current = self.q0
        j = i
        accepted = False
        word = ""
        start_index = j
        aux = ""

        while j < len(text):
            symbol = text[j]
```

```
            if current == 160 and symbol != '"':
                word += symbol
                j += 1
                column += 1
                if symbol == "\n":
                    row += 1
                    column = 1
                continue
            elif current == 164 and symbol != '\n':
                j += 1
                column += 1
                continue

            if symbol == "\n":
                symbol = "\\n"
            elif symbol == " ":
                symbol = "\\s"
            elif re.match(r'\s', symbol):
                j += 1
                continue

            if symbol.isdigit():
                symbol = int(symbol)

            if symbol not in self.sigma:
                error = Error("Lexical", "Unrecognized character '%s'" %symbol,
row, column, j, 1)
                errors.append(error)
                j += 1
                column += 1
                continue

            current =
self.matrix[self.Q.index(current)][self.sigma.index(symbol)]

            if current in self.F:
                accepted = True
                aux = symbol
                break

            j += 1
            column += 1
            word += str(symbol)

        i = j

        if accepted or j == len(text):
            if j == len(text):
```

```
                current =
self.matrix[self.Q.index(current)][self.sigma.index("\\n")]

            if current == 998:
                error = Error("Lexical", "Invalid number literal '%s'" % (word +
aux), row, column - len(word), start_index, i - start_index)
                errors.append(error)
                i += 1
                continue
            elif current == 100000 or current == 9000 or current == 163:
                if current == 9000:
                    row += 1
                    column = 1
                continue

            if current not in self.F:
                continue
            token = Token(current, word, row, column - len(word), start_index, i
- start_index)
            list.append(token)

    list.append(Token(1, 'EOF', -1, -1, -1, -1))
    return list, errors
```

6.2 Analizador sintáctico

El analizador sintáctico tiene una función para analizar cada nodo definido en la gramática libre de contexto, sin embargo tiene otras funciones que son las encargadas de detectar los errores y hacer el match para confirmar la sintaxis de una sentencia.

Primero, se debe mostrar cómo es que se va creando el árbol sintáctico, este se construye a partir de los nodos que se pre-definieron como estructuras sintácticamente válidas del lenguaje, la conexión entre nodos depende del tipo de nodo, por ejemplo un nodo si, tiene conexión con un nodo que es su valor sino, entonces, condición. El nodo raíz es el nodo programa, y se va construyendo en el análisis sintáctico iterando con un contador puntero sobre la lista de tokens y buscando estructuras válidas, estas estructuras están definidas dentro de cada método en forma secuencia y si existe más estructuras que se parecen se distinguen por ifs o excepciones para backtracking.

```
class Nodo:
    pass

class Programa(Nodo):
    def __init__(self, sentencias):
```

```
        self.sentencias = sentencias

class SentenciaSi(Nodo):
    def __init__(self, condicion, entonces, sino=None):
        self.condicion = condicion
        self.entonces = entonces  # lista de sentencias
        self.sino = sino  # lista de sentencias o None
```

En el código a continuación muestra los métodos principales del parser, current permite ver el token contenido en la posición del puntero, match es quien verifica que el token corresponda con el esperado en la estructura en caso de que no haga match se detecta error y entra en modo panic, o sea que va a continuar con el análisis ignorando los tokens que vienen hasta encontrar un token de sincronización que en nuestro caso son punto y coma o corchete. En el método parse se puede observar un ejemplo de como se define la estructura.

```
def current(self):
    if self.pos < len(self.tokens):
        return self.tokens[self.pos]
    else:
        return Token(EOF, '', -1, -1, -1, -1)

def match(self, expected_type, soft = False):
    if self.current().type == expected_type:
        self.pos += 1
    elif self.current().type == EOF:
        self.error(f"Se esperaba {expected_type}, pero se alcanzó EOF.")
        raise Exception("Fin del archivo inesperado")
    else:
        if soft:
            return
        self.error(f"Se esperaba tipo {expected_type}, se encontró
{self.current().type}")

def error(self, mensaje):
    #print(f"[Error] {mensaje} en token {self.current().type} (posición
{self.pos})")
    self.errors.append(Error("Syntax", f"[Error] {mensaje} en token
{self.current().type} (posición {self.pos})", -1, -1, self.pos, -1))
    self.panic()

def panic(self):
    if self.current().type == EOF:
        return
    while self.current().type not in self.sync_tokens and self.current().type !=
EOF:
        self.pos += 1
    if self.current().type in self.sync_tokens and self.current().type !=
RBRACE:
```

```
        self.pos += 1

def parse(self):
    try:
        if self.current().type == SPAWNEAR:
            self.match(SPAWNEAR)
            self.match(LBRACE)
            sentencias = self.lista_sentencias()
            self.match(RBRACE)
            self.match(MORIR)
            self.match(SEMICOLON)
            ret = Programa(sentencias)
            ret.index = self.pos - 1
            return ret
        else:
            self.errors.append(Error("Syntax", f"Se esperaba spawnear al
inicio", -1, -1, self.pos, -1))
            return Programa([])
    except Exception as e:
        #print(f"[Fatal] {e}")
        self.errors.append(Error("Syntax", f"[Fatal] {e}", -1, -1, self.pos,
-1))
        return Programa([])
```

Además en este código que sigue, mostramos cómo es que el backtracking está implementado, esto sirve en caso de que 2 estructuras se parezcan mucho y no se distingan por un carácter en especial, trate de hacer match con una, si falla se regresa y hace match con el otro, en este caso primero revisa si es asignación, si no lo es, entonces es expresión.

```
def sentencia(self):
    tipo = self.current().type
    if tipo == PORTAL:
        return self.declaracion_funcion()
    elif tipo in [BLOQUE, LOSA, PALANCA, LIBRO, HOJA, COFRE, ITEM]:
        return self.sentencia_declaracion()
    elif tipo == ID:
        pos_inicial = self.pos
        try:
            return self.sentencia_asignacion()
        except Exception:
            self.pos = pos_inicial
            expr = self.expresion()
            self.match(SEMICOLON)
            ret = SentenciaExpresion(expr)
            ret.index = self.pos - 1
            return ret
    elif tipo == SI:
        return self.sentencia_si()
    elif tipo == MIENTRAS:
```

```
        return self.sentencia_mientras()
    elif tipo == PARA:
        return self.sentencia_para()
    elif tipo == TELETRANSPORTAR:
        return self.sentencia_tp()
    elif tipo == SEMICOLON:
        self.match(SEMICOLON)
        ret = SentenciaVacia()
        ret.index = self.pos - 1
        return ret
    else:
        expr = self.expresion()
        self.match(SEMICOLON)
        ret =  SentenciaExpresion(expr)
        ret.index = self.pos - 1
        return ret
```

6.3 Analizador semántico

En esta fase generamos una tabla de símbolos scope-dependiente para verificar el significado del código. Aquí se suele enriquecer el AST, sin embargo por la naturaleza de nuestro proyecto (al final se traduce a alto nivel no bajo nivel) esto no es necesario y podemos trabajar con el mismo AST con alguna consideración más. A continuación se muestra la clase símbolo que construye la tabla.

```
class Simbolo:
    def __init__(self, nombre, tipo, ambito, es_funcion=False, parametros=None,
tipo_retorno=None, tipo_contenido=None):
        self.nombre = nombre
        self.tipo = tipo
        self.ambito = ambito
        self.es_funcion = es_funcion
        self.parametros = parametros or []
        self.tipo_retorno = tipo_retorno
        self.tipo_contenido = tipo_contenido
        #self.nodo = nodo tal vez usarlo, luego vemos
```

El siguiente código muestra cómo se maneja lo del scope para verificar, es una lista de diccionarios, de esta manera se van creando los símbolos dentro de un scope, cuando se sale de ese scope se hace pop y así la verificación al buscar un símbolo en un entorno solo se en el propio scope o en superiores. Una nota es que se permite el shadowing.

```
class Entorno:
    def __init__(self):
        self.ambitos = [{}]
        self.historial = []
```

```
    def entrar_ambito(self):
        self.ambitos.append({})

    def salir_ambito(self):
        self.ambitos.pop()

    def declarar(self, nombre, simbolo):
        self.ambitos[-1][nombre] = simbolo
        self.historial.append((len(self.ambitos) - 1, simbolo))  # Guarda el
nivel y el símbolo

    def buscar(self, nombre):
        for ambito in reversed(self.ambitos):
            if nombre in ambito:
                return ambito[nombre]
        return None

    def existe_en_actual(self, nombre):
        return nombre in self.ambitos[-1]
```

Ahora se muestran los principales métodos del analizador semántico para explicar su funcionamiento, funciona a través del método analizar que visita el método según el tipo de AST node que es, cada uno de estos método visitar realiza las verificaciones según el tipo de estructura que es, si coincide continua, si no continua pero marca el error. Un método muy importante es el de comparar tipos, ya que este permite regresar booleanos según dos tipos mandados para verificar su compatibilidad.

```
class AnalizadorSemantico:
    def __init__(self, tokens):
        self.entorno = Entorno()
        self.funcion_actual = None
        self.errores = []
        self.tokens = tokens

    def analizar(self, nodo):
        metodo = f"visitar_{type(nodo).__name__}"
        if(type(nodo).__name__ == "ExpresionCadena"):
            if(len(nodo.texto) == 3):
                metodo = f"visitar_ExpresionCaracter"
        elif(type(nodo).__name__[0:7] == 'Funcion'):
            metodo = f"visitar_ExpresionLlamadaFuncion"
        visitador = getattr(self, metodo, self.visitar_desconocido)
        return visitador(nodo)

    def visitar_SentenciaVacia(self, nodo):
        pass

    def visitar_desconocido(self, nodo):
```

```
        self.errores.append(Error("SEMANTICAL", f"[Error] Nodo no reconocido:
{type(nodo).__name__}", self.tokens[nodo.index].row, -1, -1, -1))

    def visitar_Programa(self, nodo):
        self.entorno.entrar_ambito()
        for sentencia in nodo.sentencias:
            self.analizar(sentencia)
        self.entorno.salir_ambito()

    def comparar_tipos(self, tipo_destino, tipo_origen):
        if tipo_destino == 'funcion' or tipo_origen == 'funcion':
            return False
        if tipo_destino == tipo_origen:
            return True
        if tipo_origen is None:
            return False
        if tipo_destino == 'item' or tipo_origen == 'item':
            return True
        if tipo_destino == 'libro' and tipo_origen == 'hoja':
            return True
        if tipo_destino == 'losa' and tipo_origen == 'bloque':
            return True
        return False
```

Un ejemplo de cómo se hace la verificación en una estructura, en este primer caso se verifica en una sentencia de declaración que contiene una lista, para esto se verifica que el tipo sea cofre o item, si no se marca error.

```
if simbolo.tipo not in ['cofre', 'item']:
    err_val = f"[Error] Variable '{nodo.nombre}' no es un cofre y se intenta
indexar"
    self.errores.append(
        Error("SEMANTICAL", err_val, self.tokens[nodo.index].row,
              -1, -1, -1))
tipo_indice = self.analizar(nodo.indice)
```

En este otro ejemplo, en una expresión binaria, si el operador es de comparación, verifica que ambos nodos hijos sean compatibles.

```
elif nodo.operador in ['==']:
    if self.comparar_tipos(tipo_izq, tipo_der):
        return 'palanca'
    else:
        err_val = f"[Error] Comparación de igualdad inválida entre '{tipo_izq}'
y '{tipo_der}'"
        self.errores.append(
            Error("SEMANTICAL", err_val, self.tokens[nodo.index].row,
                  -1, -1, -1))
```

```
        return 'item'
```

6.4 Código intermedio

El objetivo que cumple esta clase en nuestro código es pasar la construcción de alto nivel a algo más concreto (para supuestamente pasarlo a bajo nivel, aunque en este caso se regresa a alto nivel). Nuestro código intermedio es en forma de cuádruplos, siempre inicia con un OPCODE seguido del destino o nombre, seguido de esto puede o no venir los operadores necesarios.

Los métodos principales se muestran a continuación, emitir cuando es llamado añade la instrucción generada a la lista de instrucciones intermedia, generar es como un router que según el tipo de nodo AST lo lleva a su método generador correspondiente. Adicionalmente hay dos métodos auxiliares, nuevo_temp y nueva_etiqueta, el primero es importante debido a que siempre se coloca el destino de una operación, y solo se representan como binarias, entonces los temporales permiten conectar las diferentes operaciones; la siguiente es útil para distinguir las estructuras dentro del código, como donde inicia y termina if, for, while. Finalmente se coloca un método general para una asignación, primero se genera lo del lado derecho, y se emite la instrucción con OPCODE assign, el destino y como operador el valor generado.

```
class GeneradorIntermedio:
    def __init__(self):
        self.instrucciones = []
        self.temp_id = 0
        self.entorno = Entorno()
        self.etiquetas_funciones = {}
        self.contador_etiquetas = 0
        self.offset_local_actual = 0

    def nueva_etiqueta(self, prefijo="L"):
        etiqueta = f"{prefijo}{self.contador_etiquetas}"
        self.contador_etiquetas += 1
        return etiqueta

    def nuevo_temp(self):
        temp = f"t{self.temp_id}"
        self.temp_id += 1
        return temp

    def emitir(self, instruccion):
        self.instrucciones.append(instruccion)

    def generar(self, nodo):
        metodo = f"generar_{type(nodo).__name__}"
        return getattr(self, metodo, self.generar_desconocido)(nodo)
```

```
    def generar_Programa(self, nodo):
        for sentencia in nodo.sentencias:
            self.generar(sentencia)

    def generar_desconocido(self, nodo):
        raise Exception(f"No se puede generar código para
{type(nodo).__name__}")

    def generar_SentenciaVacia(self, nodo):
        pass

    def generar_SentenciaAsignacion(self, nodo):
        valor_temp = self.generar(nodo.valor)
        if nodo.es_acceso:
            indice = self.generar(nodo.indice)
            self.emitir(f"SET_LIST_ITEM MC_{nodo.nombre}, {indice},
{valor_temp}")
            #self.emitir(f"SET_LIST_ITEM {nodo.nombre}, {indice}, {valor_temp}")
        else:
            self.emitir(f"ASSIGN MC_{nodo.nombre} = {valor_temp}")
            #self.emitir(f"ASSIGN {nodo.nombre} = {valor_temp}")
```

6.5 Tipos de optimización

Después de la generación de código intermedio se aplican 3 funciones de optimización, eliminar redundancia en asignaciones, plegar constantes y eliminar código muerto. Siendo los dos primeros una optimización de mirilla dado que se hace sobre una instrucción o un bloque pequeño de código. Por otro lado, la eliminación de código muerto es una optimización global dado que se revisa todo el código para verificar que variables se usan.

La función de eliminar redundancia en asignaciones revisa instrucción por instrucción en el código intermedio y si es una ASSIGN y es una asignación redundante como x = x, la elimina en el código optimizado.

```
def _eliminar_asignaciones_redundantes(self):
    cambios = False
    nuevo_codigo = []

    for instruccion in self.codigo_intermedio:
        parts = instruccion.split(" ", 1)
        opcode = parts[0]
        args = parts[1] if len(parts) > 1 else ""
        match_assign = re.match(r'([a-zA-Z_]\w*)\s*=\s*([a-zA-Z_]\w*)', args)

        if opcode == "ASSIGN" and match_assign:
            dest_var = match_assign.group(1).strip()
            src_var = match_assign.group(2).strip()
```

```
            if dest_var == src_var:
                cambios = True
                continue
        nuevo_codigo.append(instruccion)

    if cambios:
        self.codigo_intermedio = nuevo_codigo
    return cambios
```

La función de plegado va revisando instrucción por instrucción en el código intermedio y si es que se puede calcular las constantes y sea en operaciones unarias o binarias.

```
def _plegar_constantes(self):
    cambios = False
    instrucciones_nuevas = []
    valores_conocidos = {}

    for instruccion in self.codigo_intermedio:
        if instruccion[0] == "#":
            #instrucciones_nuevas.append(instruccion)
            continue
        partes = instruccion.split(" ", 1)
        opcode = partes[0]
        args = partes[1] if len(partes) > 1 else ""

        if "=" in args:
            destino, operacion = args.split("=", 1)
            destino = destino.strip()
            operacion = operacion.strip()

            if opcode in ["ADD", "SUB", "MUL", "DIV", "MOD", "POW", "AND", "OR",
"NEG", "NOT", "EQ", "GT", "GTE", "LT", "LTE", "CON"]:
                try:
                    if opcode in ["NEG", "NOT"]:
                        op1_str = operacion
                        op1_val = valores_conocidos.get(op1_str, None)
                        if op1_val is None:
                            op1_val = float(op1_str) if '.' in op1_str else
int(op1_str)

                        resultado = None
                        if opcode == "NEG":
                            resultado = -op1_val
                        elif opcode == "NOT":
                            resultado = int(not bool(op1_val))

                        if resultado is not None:
                            instrucciones_nuevas.append(f"ASSIGN {destino} =
{resultado}")
```

```
                            valores_conocidos[destino] = resultado
                            cambios = True
                            continue
                    else:
                        op_partes = operacion.split(', ', 1)
                        if len(op_partes) != 2:
                            pass

                        op1_str, op2_str = op_partes[0].strip(),
op_partes[1].strip()
                        op1_val = valores_conocidos.get(op1_str, None)
                        if op1_val is None:
                            if opcode != "CON":
                                try:
                                    op1_val = float(op1_str) if '.' in op1_str
else int(op1_str)
                                except ValueError:
                                    pass
                            else:
                                op1_val = op1_str

                        op2_val = valores_conocidos.get(op2_str, None)
                        if op2_val is None:
                            if opcode != "CON":
                                try:
                                    op2_val = float(op2_str) if '.' in op2_str
else int(op2_str)
                                except ValueError:
                                    pass  #
                            else:
                                op2_val = op2_str
                        # Sólo plegar si AMBOS operandos son literales
                        if self._es_literal(op1_str) and
self._es_literal(op2_str):
                            a = float(op1_str) if '.' in op1_str else
int(op1_str)
                            b = float(op2_str) if '.' in op2_str else
int(op2_str)
                            resultado = None
                            if opcode == "ADD":
                                resultado = a + b
                            elif opcode == "SUB":
                                resultado = a - b
                            elif opcode == "MUL":
                                resultado = a * b
                            elif opcode == "DIV" and b != 0:
                                resultado = a / b
                            elif opcode == "MOD" and b != 0:
                                resultado = a % b
```

```
                            elif opcode == "POW":
                                resultado = a ** b
                            elif opcode == "AND":
                                resultado = int(bool(a) and bool(b))
                            elif opcode == "OR":
                                resultado = int(bool(a) or bool(b))
                            elif opcode == "EQ":
                                resultado = int(a == b)
                            elif opcode == "GT":
                                resultado = int(a > b)
                            elif opcode == "GTE":
                                resultado = int(a >= b)
                            elif opcode == "LT":
                                resultado = int(a < b)
                            elif opcode == "LTE":
                                resultado = int(a <= b)
                            elif opcode == "CON" and op1_str.startswith('"') and
op2_str.startswith('"'):
                                # Cadena literal
                                resultado = f'"{op1_str[1:-1] + op2_str[1:-1]}"'
                            if resultado is not None:
                                instrucciones_nuevas.append(f"ASSIGN {destino} =
{resultado}")
                                cambios = True
                                continue

                except ValueError:
                    pass

        instrucciones_nuevas.append(instruccion)

    self.codigo_intermedio = instrucciones_nuevas
    return cambios
```

El método de eliminación de código muerto realiza una recursión inversa y va analizando los valores que tengan las variables y guardandolos en una pila, y si es que ese valor fue usado o es una asignación que no tiene sentido realizar.

6.6 Costos de ejecución

Después de hacer la traducción, documentada en el punto 6.7, viene la ejecución que funciona de manera resumida haciendo uso de hilos separados que se se ejecutan de manera asincrónica pero al mismo tiempo necesitan estar sincronizados para el manejo de inputs y outputs dentro del programa, por lo que se utiliza un ID de sesión que identifica el código o archivo generado y así poder ejecutarlo tanto en el frontend (usando JavaScript con un webSocket) tanto en el backend con FastAPI donde se prepara el request específico que posteriormente llamará al método asíncrono del backend llamado _execute_with_session que

como ya se mencionó anteriormente, hace uso de este ID de sesión para identificar y trabajar con el mismo archivo de python (código objeto) generado.

El programa puede ejecutarse con un mínimo de 1 núcleo de CPU y 128 MB de RAM, pero estos recursos no son recomendados para un rendimiento óptimo, ya que podrían saturarse al procesar código complejo o archivos grandes.

Configuración recomendada:

- Procesador: 2 núcleos (para un análisis más fluido).
- RAM: 1 GB (evita cuellos de botella en operaciones intensivas).
- Python: Versión 3.8 o superior (requerimiento obligatorio).

Nota:
El consumo real del software no superará estos límites, pero el sistema operativo y otros procesos en segundo plano pueden aumentar los requisitos generales del sistema.

6.7 Traducción

El proceso de traducción es diferente ya que debemos regresar a un lenguaje de alto nivel, y en la generación de código intermedio se pierde abstracción, al ser python se deben considerar cosas como indentación, etc. y por lo tanto, para redibujar estructuras como funciones, ifs, ciclos, el programa en general se requieren de métodos robustos aparte, a continuación solo se muestra cómo se traducen instrucciones que se permiten hacer en una sola línea y que se identifican por el OPCODE, exiten muchos más métodos que son requeridos para poder representar la plenitud del código de MCScript en python.

```
if opcode == "ASSIGN":
   dest, src = [s.strip() for s in args.split("=", 1)]
   val = _translate_operand(src)
   translated_line = f"{dest} = {val}"
elif opcode == "ADD":
   dest, ops = [s.strip() for s in args.split("=", 1)]
   op1, op2 = [s.strip() for s in ops.split(",")]
    translated_line = f"{dest} = weak_arithmetic({_translate_operand(op1)}) +
weak_arithmetic({_translate_operand(op2)})"
elif opcode == "SUB":
   dest, ops = [s.strip() for s in args.split("=", 1)]
   op1, op2 = [s.strip() for s in ops.split(",")]
    translated_line = f"{dest} = weak_arithmetic({_translate_operand(op1)}) -
weak_arithmetic({_translate_operand(op2)})"
elif opcode == "MUL":
   dest, ops = [s.strip() for s in args.split("=", 1)]
   op1, op2 = [s.strip() for s in ops.split(",")]
    translated_line = f"{dest} = weak_arithmetic({_translate_operand(op1)}) *
weak_arithmetic({_translate_operand(op2)})"
elif opcode == "DIV":
```

```
    dest, ops = [s.strip() for s in args.split("=", 1)]
    op1, op2 = [s.strip() for s in ops.split(",")]
     translated_line = f"{dest} = weak_arithmetic({_translate_operand(op1)}) /
weak_arithmetic({_translate_operand(op2)})"
elif opcode == "POW":
    dest, ops = [s.strip() for s in args.split("=", 1)]
    op1, op2 = [s.strip() for s in ops.split(",")]
     translated_line = f"{dest} = weak_arithmetic({_translate_operand(op1)}) **
weak_arithmetic({_translate_operand(op2)})"
elif opcode == "MOD":
    dest, ops = [s.strip() for s in args.split("=", 1)]
    op1, op2 = [s.strip() for s in ops.split(",")]
     translated_line = f"{dest} = weak_arithmetic({_translate_operand(op1)}) %
weak_arithmetic({_translate_operand(op2)})"
elif opcode == "NEG":
    dest, ops = [s.strip() for s in args.split("=", 1)]
    op1 = ops.strip()
                    translated_line      =      f"{dest}      =      (-1      *
weak_arithmetic({_translate_operand(op1)}))"
elif opcode in ["EQ", "LT", "GT", "LTE", "GTE"]:
    dest, ops = [s.strip() for s in args.split("=", 1)]
    op1, op2 = [s.strip() for s in ops.split(",")]
    op_map = {"EQ": "==", "LT": "<", "GT": ">", "LTE": "<=", "GTE": ">="}
      translated_line = f"{dest} = {_translate_operand(op1)} {op_map[opcode]}
{_translate_operand(op2)}"
elif opcode == "AND":
    dest, ops = [s.strip() for s in args.split("=", 1)]
    op1, op2 = [s.strip() for s in ops.split(",")]
          translated_line   =   f"{dest}   =   {_translate_operand(op1)}   and
{_translate_operand(op2)}"
elif opcode == "OR":
    dest, ops = [s.strip() for s in args.split("=", 1)]
    op1, op2 = [s.strip() for s in ops.split(",")]
          translated_line   =   f"{dest}   =   {_translate_operand(op1)}   or
{_translate_operand(op2)}"
elif opcode == "CON":
    dest, ops = [s.strip() for s in args.split("=", 1)]
    op1, op2 = [s.strip() for s in ops.split(",")]
         translated_line   =   f"{dest}   =   str({_translate_operand(op1)})   +
str({_translate_operand(op2)})"
elif opcode == "NOT":
    dest, op = [s.strip() for s in args.split("=", 1)]
    translated_line = f"{dest} = not {_translate_operand(op)}"
elif opcode == "PRINT":
    translated_line = f"await async_print({_translate_operand(args)})"
elif opcode == "INPUT":
    temp, message = args.split(', ')
    translated_line = f"{temp} = await async_input({message})"
```

Se hace la nota, que al traducir se añaden ciertos métodos que son importantes para nosotros que permiten que la ejecución en el navegador web sea correcta y compatible con inputs, y que la naturaleza descrita de item este presente, por ejemplo:

```
def weak_arithmetic(x):
   try: return float(x)
   except: return len(x)
```

```
async def async_input(prompt=""):
   global _execution_session
   if _execution_session:
       return await _execution_session.wait_for_input(prompt)
   else:
       return input(prompt)
```

7. Interfaz del lenguaje

La interfaz se hizo con una inspiración en el videojuego de minecraft, tratando de que sea amigable con los usuarios y fácil de usar.

Imagen 2. Bienvenida a MCScript

Lo primero que ve el usuario cuando entra por primera vez, es una bienvenida con una ventana en la que se explica qué es MCScript, se colocan la definición de las palabras reservadas del lenguaje y para qué sirven. Además, se colocan algunos ejemplos en código MCScript que permite al usuario familiarizarse mejor con la lógica y ejecutar sus primeros pasos.

De igual forma, el usuario puede acceder a esta ventana mediante un botón de ayuda que aparece en la parte superior.

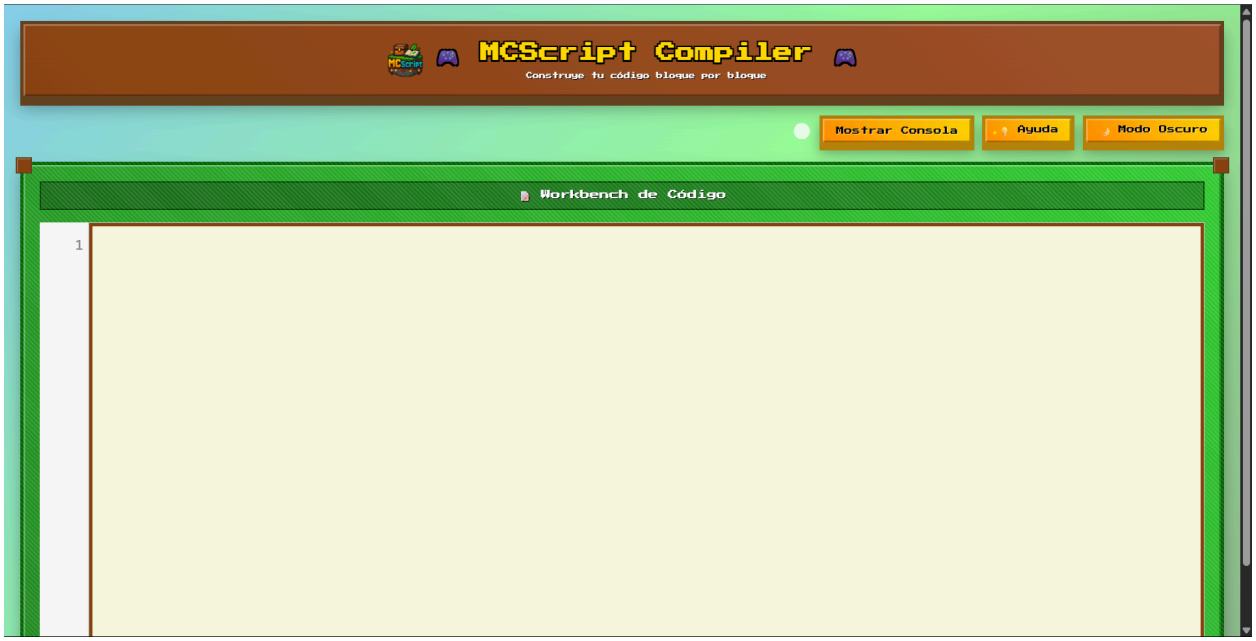

Imagen 3. Workbench de MCScript

Después de que cierra la ventana de bienvenida, el usuario tiene acceso al Workbench de código, en el cuál el usuario puede escribir código MCScript y debajo del Workbench hay un botón llamado “Compilar bloques”, que como su nombre indica debe presionar para que se haga el proceso de compilación.

Imagen 4. Vista de ejemplo de código sin errores

Si el programa se compila correctamente sin errores, adicionalmente al botón de compilar, se habilita del lado derecho un botón llamado “Ejecutar” que se encarga de ejecutar el código en MCScript, es decir, convierte el código fuente a código objeto.

Imagen 5. Pestaña de consola “Tokens”

El usuario tiene acceso también a diferentes pestañas dentro de la consola que se le proporciona. La primera de ellas es la ventana de “Tokens” que muestra el resultado del análisis léxico del compilador.

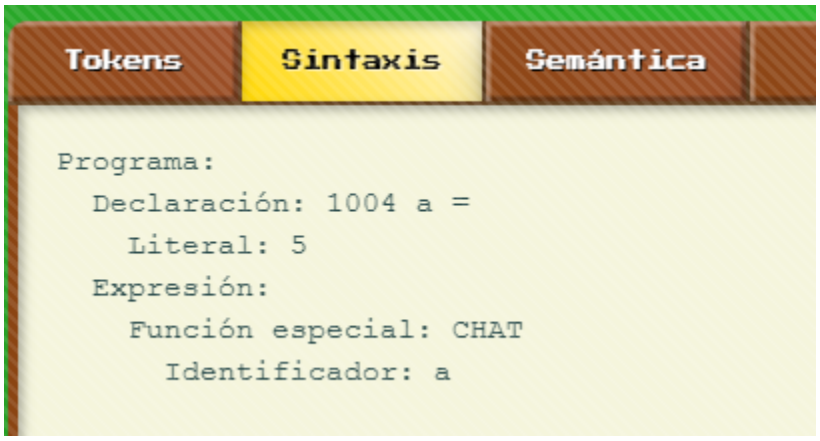

Imagen 6. Pestaña de consola “Sintaxis”

La siguiente pestaña muestra el resultado del análisis sintáctico.

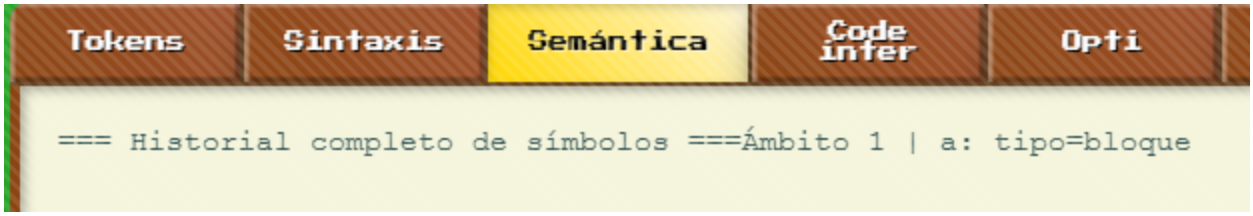

Imagen 7. Pestaña de consola “Semántica”

La pestaña de Semántica muestra el resultado del análisis semántico.

Estas pestañas principalmente, son muy útiles principalmente para depuración de errores, ya que le muestra al usuario si es que cometió algún error en las primeras fases de compilación, indicando al mismo tiempo qué token o cosa es la que se esperaba.

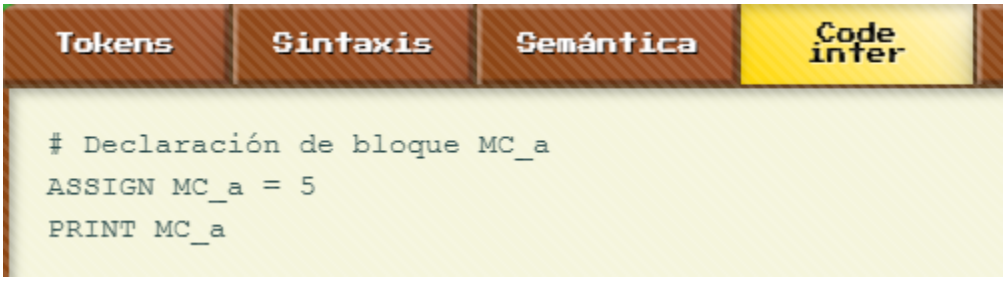

Imagen 8. Pestaña de consola “Código Intermedio”

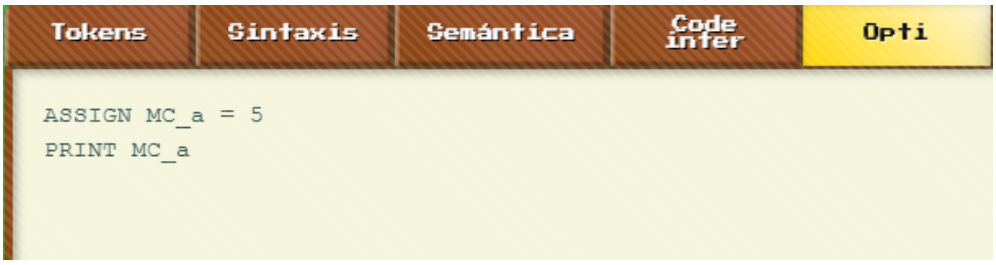

Imagen 9. Pestaña de consola “Código Optimizado”

También se tiene acceso a las pestañas de código intermedio y código optimizado que proporciona una vista de cómo es que el código escrito en MCScript se convierte a una representación intermedia que facilitará el proceso de traducción a código objeto.

Imagen 10. Pestaña de consola “Output”

Casi para finalizar, el usuario también puede visualizar la traducción generada del código que escribió en su representación de objeto, que en el caso de nuestro compilador, convierte el código MCScript a su equivalente en python para poder ser ejecutado.

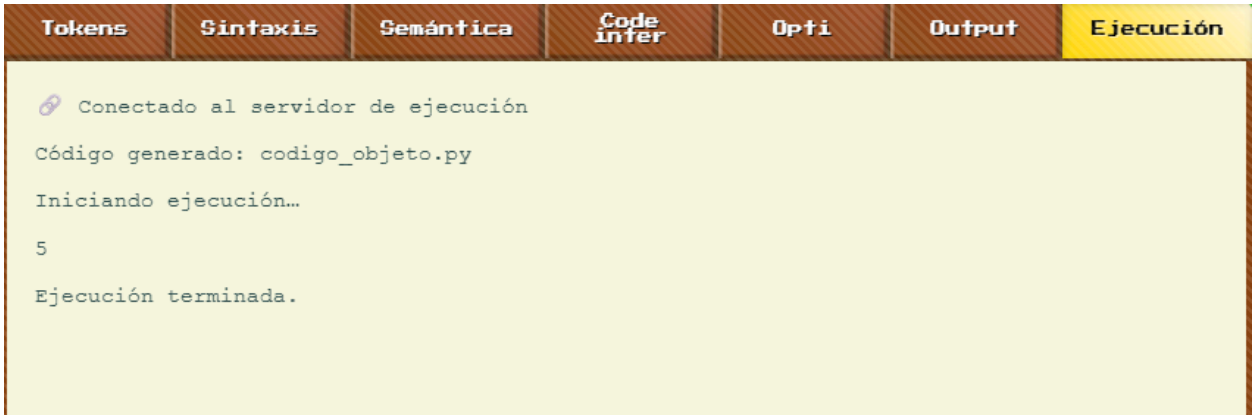

Imagen 11. Pestaña de consola “Ejecución”

La última pestaña que puede visualizar el usuario es la de “Ejecución”, en la puede ver el resultado de su código MCScript, en dado caso de que haya utilizado instrucciones como “chat” que imprime en pantalla, se visualizarán en esta pestaña. De igual manera, si el usuario utilizó instrucciones de entrada como “cartel” aparecerá un modal que le pedirá que ingrese lo que sea que haya solicitado como puede apreciarse en la siguiente imagen.

Imagen 12. Modal de lectura de input

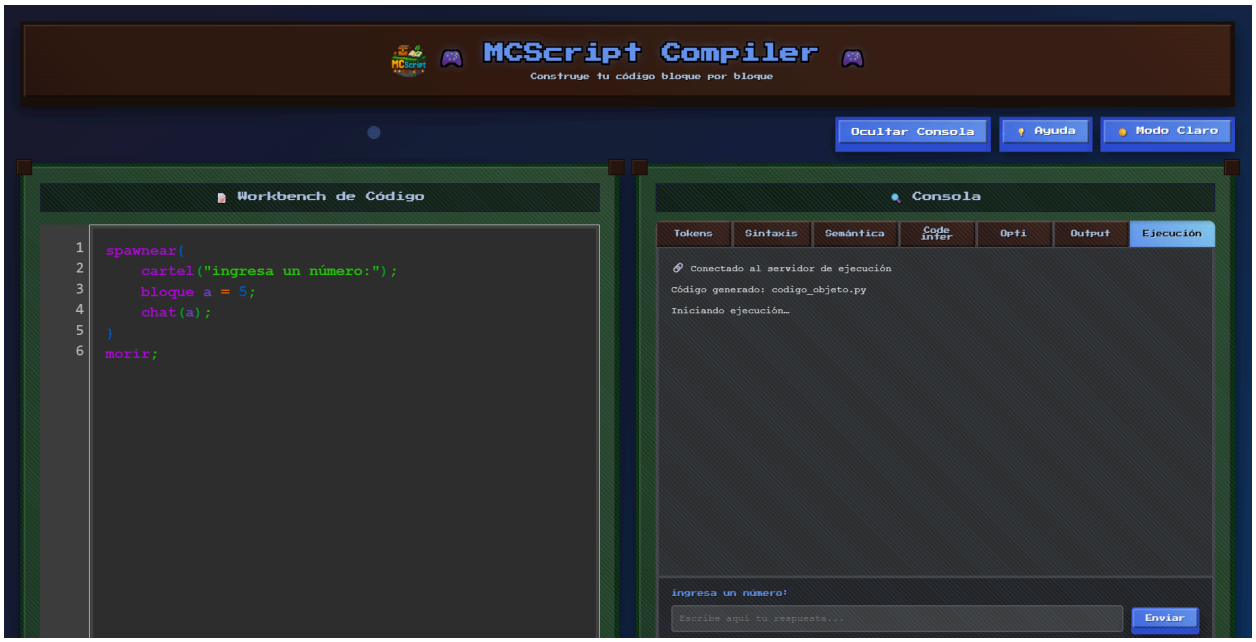

Imagen 13. Modo oscuro de MCScript

Finalmente, el usuario también puede colocar un modo oscuro para el entorno de desarrollo mediante el botón de “Modo Oscuro/Claro”.

8. Manual de usuario

Bienvenido a MCScript

🎯 ¿Qué es MCScript?
MCScript es un lenguaje de programación inspirado en Minecraft que te permite crear código de forma sencilla y divertida, usando palabras del videojuego para representar conceptos de la programación y facilitar su entendimiento.

📝 Hola Mundo en MCScript

```
spawnear {
    chat("Hola MCScript!!");
}
Morir;
```

🔧 ¿Cómo usar el compilador?
Escribir código: Deberás de escribir el código que deseas probar, puedas usar el ejemplo de arriba como inicio

Compilar: Para poder usar el código que escribiste, debes de dar click sobre el botón "Compilar" debajo de la sección de código

Ejecutar: Una vez que compilaste el código, ahora lo puedes ejecutar dando click en el botón "Ejecutar" y el resultado se mostrará en la pestaña de Ejecución de la consola

❓ NOTAS IMPORTANTES
Si modificas tu código y quieres volver a probarlo, primero debes compilar nuevamente el código y después ejecutarlo, de lo contrario volverás a ejecutar el código anterior
La Consola tiene varias pestañas en las que puedes visualizar información acerca del proceso de compilación pero la más importante y la que muestra el resultado de tu código es la de "Ejecución"

Mientras escribres tu código es normal que aparezcan subrayadas las palabras que escribes o se ilumine de color rojo la línea que escribiste, pero no te asustes, es el indicador de errores en tiempo real que tiene el compilador

🔧 Tipos de Datos
bloque: Este define todos los números enteros (ej: 42, -10)
losa: Con este podemos poner números decimales (ej: 3.14, -2.5)
hoja: Este tipo de dato solo acepta caracteres individuales (ej: "a", "b")
libro: Este otro acepta cadenas de texto completas (ej: "hola", "adios)
palanca: Este es el booleano, es decir, sólo acepta valores sí o no que son "encendido" y "apagado" respectivamente

🎮 Palabras Clave
spawnear: Esta palabra define el inicio del bloque de código
morir: Como su nombre indica, es la muerte o el fin del bloque de código
craftear: Esta palabra recibe dos parámetros que se sumaran
romper: Esta palabra recibe dos parámetros que se restaron
apilar: Esta palabra recibe dos parámetros que se multiplicaran
repartir: Esta palabra recibe dos parámetros que se dividirán

sobrar: Esta palabra recibe dos parámetros enteros que se dividirán y devolverá el sobrante de la división
encantar: Esta palabra recibe dos parámetros que el primero se elevará a la potencia que indique el segundo
chat: Esta palabra se usa para imprimir un mensaje en la consola
letrero: Esta palabra se usa para leer un valor de la consola, con un mensaje
antorchaR: Esta palabra se usa para invertir un valor booleano

🎮 Bucles
para: Sirve para repetir algo varias veces definidas, uno por uno, como si estuvieras pasando lista
mientras: Sirve para repetir algo varias veces indefinidas mientras la condición se cumpla

🎮 Condicionales
Si: Esta sentencia abre un bloque que código que se ejecutará siempre y cuando la condición se cumpla.
Sino: Esta sentencia abre un bloque que código que se ejecutará cuando la condición del Si no se cumpla, solo puede estar después del bloque del Si

9. Características del equipo para el uso del lenguaje.

El equipo del usuario no necesita tener alguna característica en especial para poder ejecutar el compilador, esto debido a que nuestro compilador está hecho como un servicio web al que se puede acceder libremente únicamente con una conexión a internet. Por lo tanto, el usuario puede ejecutar código MCScript desde cualquier dispositivo que pueda acceder a la página.

10. Ejemplos y casos de uso

💡 Ejemplo Completo

```
spawnear {
    chat("¿Hola cómo estás?");
    libro respuesta = cartel("ingresa tu respuesta:");
    //imprimimos respuesta
    chat(respuesta);
}
morir;
```

Tokens identificados:
1023, 4001, 1026, 5001, 3004, 5002, 3003, 1011, 6000, 2071, 1027, 5001, 3004, 5002, 3003, 1026, 5001, 6000, 5002, 3003, 4002, 1013, 3003, 1

Sintaxis:
Programa:
  Expresión:
    Función especial: CHAT
      Cadena: "¿Hola cómo estás?"
  Declaración: 1011 respuesta =
    Desconocido: <class 'Compiler.ast_nodes.FuncionCartel'>
  Expresión:
    Función especial: CHAT
      Identificador: respuesta

Semantica:
=== Historial completo de símbolos ===Ámbito 1 | respuesta: tipo=libro

Codigo Intermedio:
PRINT "¿Hola cómo estás?"
Declaración de libro MC_respuesta
INPUT t0, "ingresa tu respuesta:"
ASSIGN MC_respuesta = t0
PRINT MC_respuesta

Codigo intermedio optimizado:
PRINT "¿Hola cómo estás?"
INPUT t0, "ingresa tu respuesta:"
ASSIGN MC_respuesta = t0
PRINT MC_respuesta

```
Iniciando ejecución…
¿Hola cómo estás?
```

11. Por qué vale la pena adquirir mi lenguaje.

MCScript es un lenguaje de programación diseñado con el objetivo principal de introducir a niños y principiantes en el mundo de la programación, utilizando un enfoque gamificado inspirado en el popular videojuego Minecraft. A continuación, se detallan las razones por las que vale la pena adquirirlo y trabajar con él:

Facilita el Aprendizaje
MCScript busca simplificar la curva de aprendizaje al abstraer conceptos complejos de programación con equivalentes del universo Minecraft. Esto permite que los usuarios se familiaricen con la lógica de programación de una manera más amena y sencilla, haciendo la transición a otros lenguajes de alto nivel como java o python más fluida.

Características Clave

- Sintaxis sencilla: Se basa en Python, incorporando elementos comunes de otros lenguajes como el uso de corchetes, punto y coma, y un tipado fuerte.

- Conceptos relacionados con Minecraft: Conecta ideas fundamentales de programación con elementos del juego, como la concatenación de strings que se visualiza como la unión de hojas en un libro.

- Soporte esencial: Incluye soporte para estructuras de control básicas como condiciones (SI, SINO), bucles (MIENTRAS, PARA), funciones (PORTAL, TELETRANSPORTAR) y la estructura de datos de listas (llamadas cofres).

- Interfaz de usuario atractiva: Ofrece una interfaz intuitiva y llamativa que demuestra cada fase del proceso de compilación, desde el análisis léxico hasta la generación de código.

Impacto Social y Accesibilidad
MCScript tiene un potencial social significativo al combinar la programación con un videojuego tan conocido como Minecraft. Al ser un lenguaje simple, accesible, educativo y gamificado, puede despertar el interés por la tecnología en comunidades que quizás no se sentirían atraídas por lenguajes de programación tradicionales.

12. Conclusiones.

El desarrollo de MCScript representa un avance significativo en la gamificación del aprendizaje de la programación, al integrar conceptos fundamentales de la informática con el universo inmersivo de Minecraft. Este proyecto no solo demuestra la viabilidad de los Lenguajes de Dominio Específico (DSL) en entornos educativos, sino que también subraya la importancia de adaptar las metodologías de enseñanza a los intereses de las nuevas generaciones.

A lo largo de este documento, se ha detallado meticulosamente cada fase de la construcción del compilador , desde el análisis léxico que transforma el código fuente en tokens , pasando por el análisis sintáctico que valida la estructura gramatical mediante Gramáticas Libres de Contexto (GLC) y genera el Árbol de Sintaxis Abstracta (AST) , hasta el análisis semántico que asegura la coherencia y el significado del programa con el soporte de la tabla de símbolos. Asimismo, se ha abordado la generación de código intermedio , las estrategias de optimización

para mejorar la eficiencia del código , y finalmente, la generación del código objeto. La elección de Python como lenguaje de implementación y el uso de FastAPI, Jinja2, HTML, CSS y JavaScript para la interfaz web, resaltan un enfoque tecnológico moderno y modular que facilita la accesibilidad y la interactividad del compilador.

Si bien MCScript está limitado en su aplicación para el desarrollo de software complejo del mundo real debido a su diseño educativo y la ausencia de una biblioteca estándar o mecanismos avanzados de gestión de errores , su verdadero valor reside en su potencial como herramienta pedagógica. Ofrece una puerta de entrada accesible y estimulante a la programación, permitiendo a los principiantes "construir su código, bloque a bloque" y fomentando el pensamiento computacional de una manera lógica. MCScript demuestra que los lenguajes de programación pueden ser divertidos y educativos, abriendo nuevas vías para la enseñanza de las ciencias de la computación.

Referencias

[1] Aho, A. V., Lam, M. S., Sethi, R., & Ullman, J. D. (2007). Compilers: Principles, Techniques, and Tools (2nd ed.). Pearson.

[2] Grune, D., Bal, H. E., Jacobs, C. J. H., & Langendoen, K. (2012). Modern Compiler Design (2nd ed.). Springer.

[3] Muchnick, S. S. (1997). Advanced Compiler Design and Implementation. Morgan Kaufmann.

[4] Hopcroft, J. E., Motwani, R., & Ullman, J. D. (2007). *Introduction to Automata Theory, Languages, and Computation* (3rd ed.). Pearson.

[5] Scott, M. L. (2009). *Programming Language Pragmatics* (3rd ed.). Morgan Kaufmann.

[6] Fowler, M. (2010). *Domain-Specific Languages*. Addison-Wesley.

[7] Mernik, M., Heering, J., & Sloane, A. M. (2005). When and How to Develop Domain-Specific Languages. *ACM Computing Surveys*, 37(4), 316–344.

[8] Van Deursen, A., Klint, P., & Visser, J. (2000). Domain-Specific Languages: An Annotated Bibliography. SIGPLAN Notices, 35(6), 26–36.

[9] Short, D. (2012). Teaching Scientific Concepts Using a Virtual World—Minecraft. Teaching Science, 58(3), 55–58.4

[10] Clark, M., & Bower, M. (2021). Minecraft for Education: A Qualitative Study of Teacher and Student Experiences in the Classroom. *British Journal of Educational Technology*, 52(3), 1072–1087. https://doi.org/10.1111/bjet.13081

[11] Van Rossum, G., & Drake, F. L. (2009). *The Python Language Reference Manual* (Release 3.0.1). Network Theory Ltd.

[12] Lutz, M. (2021). *Programming Python* (5th ed.). O'Reilly Media.

[13] Ramírez, S. (2019). *FastAPI Documentation*. https://fastapi.tiangolo.com/

[14] Ronacher, A. (2008). *Jinja2 Documentation*. https://jinja.palletsprojects.com/

[15] Flanagan, D. (2020). *JavaScript: The Definitive Guide* (7th ed.). O'Reilly Media.

[16] Duckett, J. (2011). *HTML & CSS: Design and Build Websites*. John Wiley & Sons.