TEC | Tecnológico de Costa Rica

Instituto Tecnológico de Costa Rica

Escuela de Computación

Compiladores e Intérpretes, IC5701

Etapa Cero: Definición del lenguaje

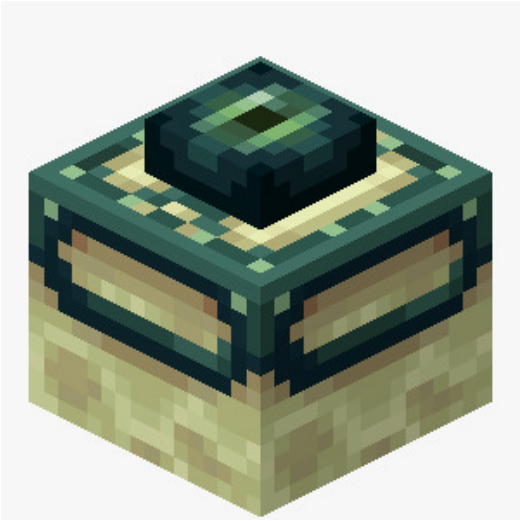
Estudiantes:
Samir Cabrera, 2022161229
Luis Urbina, 2023156802

Primer semestre del año 2025

Índice general

0.1. Concurso de Nombres

EnderLang



Justificación: EnderLang es un lenguaje de programación inspirado en la dimensión más misteriosa y poderosa de Minecraft: el End. En el juego, el End es un lugar enigmático, habitado por criaturas únicas como los Enderman y dominado por el poderoso Ender Dragon. Esta dimensión es inaccesible hasta que el jugador alcanza un alto nivel de habilidad y preparación, lo que simboliza el dominio de un conocimiento avanzado y profundo.

Tipo de archivo: .edlg

0.2. Definición del Lenguaje

0.2.1. Elementos Fundamentales del lenguaje

0.2.2. Estructura Básica

Sección de Constantes (Bedrock) Representa la base indestructible y fundamental. Las constantes, como el bedrock, son valores inmutables que una vez definidos no pueden ser alterados durante la ejecución del programa.

Ejemplo:

```
bedrock {
    // Declaraciones de constantes
}
```

Sección de Tipos (Shulker Box) Las shulker box son contenedores versátiles que pueden almacenar diferentes tipos de items. De manera similar, esta sección define los diferentes tipos de datos que podrán ser utilizados en el programa.

Ejemplo:

```
shulker_box {
    // Declaración de tipos
}
```

Sección de Variables (Chest) Los cofres almacenan y permiten modificar su contenido. Las variables funcionan como contenedores que pueden almacenar y actualizar valores durante la ejecución.

Ejemplo:

```
chest {
    // Declaraciones de variables
}
```

Sección de Prototipos (CraftingTable) La mesa de crafteo permite crear nuevos objetos a partir de elementos básicos. Los prototipos definen nuevas estructuras que serán utilizadas en el programa.

Ejemplo:

```
crafting_table {
    // Declaraciones de prototipos
}
```

Sección de Rutinas (RedstoneCircuit) Los circuitos de redstone pueden ejecutar algún funcionamiento en el juego. Representan bloques de código que realizan tareas determinadas.

Ejemplo:

```
redstone_circuit {
     // Declaraciones de rutinas
 }
```

Punto de Entrada (Spawn) El punto de spawn es donde comienza el jugador. Define el punto inicial de ejecución del programa.

Ejemplo:

```
spawn {
    // Código principal
}
```

Sistema de Asignación de Constantes (Beacon) El beacon emite efectos constantes y permanentes. Establece el sistema para definir valores inmutables.

Ejemplo:

```
 bedrock {
     beacon NIVEL_MAXIMO = 256;
     beacon PROFUNDIDAD_MINIMA = -64;
 }
```

Sistema de Asignación de Tipos (Anvil) El yunque modifica y combina items. Define cómo se asignan y modifican los tipos de datos.

Ejemplo:

```
anvil número -> texto;
anvil flotante -> entero;
```

Sistema de Declaración de Variables (ItemFrame) Los marcos de item muestran y organizan objetos. Establece cómo se declaran y organizan las variables.

Ejemplo:

```
chest {
     item_frame salud = 20;
     item_frame nombre = "Steve";
 }
```

0.2.3. Tipos de Datos

Tipo de Dato Entero (Emerald) Las esmeraldas representan valores discretos y contables. Utilizadas como moneda principal del juego.

Ejemplo:

```
shulker_box {
    emerald contador;
    emerald nivel = 64;
}
```

Tipo de Dato Caracter (Book) Los libros almacenan símbolos individuales del lenguaje.

Ejemplo:

```
shulker_box {
    book letra;
    book inicial = 'A';
}
```

Tipo de Dato String (BookAndQuill) El libro con pluma permite almacenar secuencias de caracteres.

Ejemplo:

```
shulker_box {
    book_and_quill mensaje;
    book_and_quill nombre = "Steve";
}
```

Tipo de Dato Booleano (RedstoneTorch) La antorcha de redstone tiene dos estados posibles, como los valores booleanos.

Ejemplo:

```
shulker_box {
    redstone_torch estaEncendido;
    redstone_torch tieneLlave = true;
}
```

Tipo de Dato Conjunto (BannerPattern) Los patrones de banderas representan colecciones de elementos únicos.

Ejemplo:

```
shulker_box {
    banner_pattern colores;
    banner_pattern herramientas = {pico, pala, hacha};
}
```

Tipo de Dato Archivo de Texto (Map) Los mapas almacenan y muestran información como los archivos de texto.

Ejemplo:

```
shulker_box {
    map registro;
    map bitacora = "log.txt";
}
```

Tipo de Dato Flotante (GoldNugget) Las pepitas de oro representan fracciones o partes de un lingote completo, similar a los números decimales.

Ejemplo:

```
chest {
    gold_nugget temperatura = 36.5;
    gold_nugget velocidad = 1.5;
}
```

Tipo de Dato Arreglo (Bundle) El saco puede almacenar varios ítems del mismo tipo, como un arreglo almacena elementos del mismo tipo.

Ejemplo:

```
chest {
    bundle[3] herramientas = ["pico", "pala", "hacha"];
}
```

Tipo de Dato Registro (Structure) Las estructuras en Minecraft son construcciones que contienen múltiples bloques diferentes organizados de manera específica, similar a cómo un registro contiene diferentes campos.

Ejemplo:

```
chest {
    structure Jugador {
        book_and_quill nombre;
        emerald nivel;
        gold_nugget salud;
```

```
    }
    structure Jugador steve;
}
```

0.2.4. Datos Creativos

A continuacion se da la propuesta de algunos datos creativos que se podran usar durante el funcionamiento del lenguaje Enderlang.

Se hace referencia a que los tipos de datos creativos tendran el prefijo .^{EG}G", esto ya que en Minecraft el item EGG es el origen de varios mobs, por lo que es una buena referencia para varios tipos de datos.

Un ejemplo del uso de EGG es el siguiente:

```
chest {
    egg TipoDeDatoCreativo nombreVariable= dato a recibir;
}
```

a continuacion se detallan los tipos de datos creativos.

Geo (Compass) El tipo de dato geo hace referencia a dos numeros almacenados en una misma unidad, esto con el fin de poder guardar coordenadas, el nombre representativo de la variable sera çompassza que esta es la brujula de minecraft, a continuacion se brinda un ejemplo de su uso.

```
chest {
    egg compass heredia = 123.55,1234.123;
}
```

Se deben de colocar dos numeros con un punto decimal separados por una coma.

Probabilidad (Spider Eye) El tipo de dato probabilidshingRodad hace referencia a un numero que este entre 0 y 1, este dato se especializa para trabajar con probabilidades, se asigna el name de "spidereye"porque en el juego de Minecraft el uso de la cana de pescar puede llegar a presentar cierta probabilidad de exito, a continuacion se brinda un ejemplo de su uso.

```
chest {
    egg spidereye probabilidad = 0.56;
}
```

Incertidumbre (Pumpkin) El tipo de dato incertidumbre hace referencia a un numero acompanado de un punto, la primera parte del numero hace referencia al numero que tiene una medicion, y despues del punto esta su incertidumbre, este dato esta pensado para calculos cientificos. Ademas se decidio que el nombre sea pumpkin ya que es uno de los items del juego con menos probabilidad de aparecer lo que genera una alta incertidumbre por partida, a continuacion se brinda un ejemplo de su uso.

```
chest {
    egg pumpkin medicion = 58.97
}
```

0.2.5. Literales

Literales Booleanas (Lever) Las palancas tienen dos estados definidos (on/off), representando los valores booleanos literales.

Ejemplo:

```
chest {
    redstone_torch estado = lever_on;  // true
    redstone_torch apagado = lever_off;  // false
}
```

Literales de Conjuntos (FireworkStar) Las estrellas de fuegos artificiales pueden tener diferentes efectos y colores, representando conjuntos de elementos.

Ejemplo:

```
chest {
    banner_pattern colores = firework_star{rojo, verde, azul};
    banner_pattern minerales = firework_star{hierro, oro, diamante};
}
```

Literales de Archivos (BookItem) Los libros pueden guardar y cargar información, similar a los archivos.

Ejemplo:

```
chest {
    map registro = book_item("bitacora.txt");
    map configuracion = book_item("config.dat");
}
```

Literales de Números Flotantes (SplashPotion) Las pociones arrojadizas tienen efectos con valores decimales de duración e intensidad.

Ejemplo:

```
chest {
    gold_nugget velocidad = splash_potion(3.14);
    gold_nugget gravedad = splash_potion(-9.81);
}
```

Literales de Enteros (Diamond) Los diamantes representan valores enteros precisos y valiosos.

Ejemplo:

```
chest {
    emerald nivel = diamond(64);
    emerald profundidad = diamond(-64);
}
```

Literales de Caracteres (NameTag) Las etiquetas de nombre contienen caracteres que identifican entidades.

Ejemplo:

```
chest {
    book direccion = name_tag('N');
    book grado = name_tag('A');
}
```

Literales de Strings (Sign) Los carteles muestran mensajes de texto completos.

Ejemplo:

```
chest {
    book_and_quill mensaje = sign("Hola Mundo");
    book_and_quill nombre = sign("Steve");
}
```

Literales de Arreglos (Minecart) Los minecarts pueden transportar diferentes items en orden, como un arreglo.

Ejemplo:

```
chest {
    bundle[4] materiales = minecart[1, 2, 3, 4];
}
```

Literales de Registros (Armor Stand) Los soportes de armadura mantienen múltiples piezas en posiciones específicas.

Ejemplo:

```
chest {
    structure Equipamiento = armor_stand{
        casco: "diamante",
        pechera: "hierro",
        nivel: 2
    };
}
```

0.2.6. Sistemas de Acceso a Datos

Sistema de Acceso Arreglos (Hopper) Similar a cómo un hopper extrae items de un cofre, este permite acceder y modificar elementos individuales de un arreglo mediante índices.

Ejemplo:

```
// Declaramos un arreglo de 5 posiciones
bundle[5] inventario;

// Escribimos el valor 64 en la primera posición del arreglo
hopper(inventario[0]) = diamond(64);

// Leemos el valor de la segunda posición
emerald item = hopper(inventario[1]);
```

Sistema de Acceso Strings (Comparator) Como un comparador de redstone mide señales específicas, este sistema permite examinar y modificar caracteres individuales dentro de una cadena de texto mediante índices.

Ejemplo:

```
// Creamos una cadena de texto
book_and_quill texto = sign("Minecraft");

// Obtenemos un carácter individual
book caracter = comparator(texto[0]);  // Obtiene 'M'

// Modificamos un carácter en la cadena
comparator(texto[5]) = name_tag('C');
```

Sistema de Acceso Registros (Observer) Como un observer detecta cambios en bloques, este sistema permite acceder y modificar campos específicos dentro de un registro usando notación de punto.

Ejemplo:

```
// Definimos y creamos un registro de tipo Equipamiento
structure Equipamiento {
    book_and_quill casco;
    emerald nivel;
    gold_nugget durabilidad;
};
structure Equipamiento equipo;

// Leemos el valor del campo 'casco'
book_and_quill material = observer(equipo.casco);

// Modificamos el valor del campo 'nivel'
observer(equipo.nivel) = diamond(3);
```

Asignación y Familia (Dispensador) Como un dispensador que distribuye diferentes tipos de items según se necesite, este sistema permite asignar valores a variables manteniendo la compatibilidad entre tipos de una misma familia.

Ejemplo:

```
// Asignación simple
emerald nivel = diamond(1);

// Asignación entre familia de números
gold_nugget experiencia = dispenser(nivel);    // entero a flotante
emerald puntos = dispenser(experiencia);       // flotante a entero
```

0.2.7. Operadores y Expresiones

Operaciones aritméticas básicas de enteros (Sword) Como una espada que suma o resta puntos de vida al golpear, las operaciones aritméticas básicas de enteros permiten sumar, restar, multiplicar o dividir números enteros en el programa.

Ejemplo:

```
// Declaramos variables enteras
emerald daño = 10;
```

```
emerald defensa = 5;

// Operaciones básicas
emerald resultado;
resultado = sword(daño + defensa);  // suma
resultado = sword(daño - defensa);  // resta
resultado = sword(daño * 2);        // multiplicación
resultado = sword(daño / 2);        // división
```

Incremento y Decremento (Arrow) Al igual que una flecha que avanza o retrocede una posición al ser disparada, el incremento y decremento ajustan el valor de una variable hacia arriba o hacia abajo en una unidad.

Ejemplo:

```
emerald flechas = 10;

// Incremento
flechas++;  // flechas = 11

// Decremento
flechas--;  // flechas = 10

// Pre-incremento y pre-decremento
emerald total = ++flechas;  // incrementa y luego asigna
total = --flechas;          // decrementa y luego asigna
```

Operaciones básicas sobre caracteres (BookShelf) Así como las estanterías organizan libros con letras y palabras, las operaciones sobre caracteres permiten manipular y transformar letras en el programa, similar a cómo ordenamos o buscamos libros en una estantería.

Ejemplo:

```
book letra = name_tag('a');

// Conversión a mayúscula usando bookshelf
book mayuscula = book_shelf(letra.toUpper());  // 'A'

// Obtener código ASCII del carácter
emerald codigo = book_shelf(letra.toCode());

// Verificar si es una letra
redstone_torch esLetra = book_shelf(letra.isLetter());
```

Operaciones lógicas solicitadas (RedstoneDust) Al igual que el polvo de redstone controla el flujo de energía con verdadero (encendido) o falso (apagado), las operaciones lógicas permiten tomar decisiones en el código basadas en condiciones.

Ejemplo:

```
redstone_torch señal1 = lever_on;   // true
redstone_torch señal2 = lever_off;  // false

// Operaciones lógicas básicas
redstone_torch resultado;
resultado = señal1 && señal2;  // AND
resultado = señal1 || señal2;  // OR
resultado = !señal1;           // NOT
```

Operaciones de Strings solicitadas (Fishing Rod) Como la caña de pescar que conecta y atrae objetos, las operaciones de strings conectan (concatenan) y manipulan cadenas de texto en el programa.

Ejemplo:

```
book_and_quill nombre = sign("Steve");
book_and_quill saludo = sign("Hola ");

// Concatenación de strings
book_and_quill mensaje = fishing_rod(saludo + nombre);
```

Operaciones de conjuntos solicitadas (Campfire) Así como la fogata combina ítems para crear humo y señales, las operaciones de conjuntos permiten unir, intersectar o diferenciar colecciones de datos.

Ejemplo:

```
banner_pattern minerales = {diamante, hierro, oro};
banner_pattern herramientas = {pico, pala, hacha};

// Unión de conjuntos
banner_pattern unidos = campfire(minerales + herramientas);

// Intersección
banner_pattern comunes = campfire(minerales * herramientas);
```

Operaciones de archivos solicitadas (Barrel) Al igual que un barril que guarda y organiza ítems, las operaciones de archivos permiten almacenar, leer y modificar datos en un archivo externo.

Ejemplo:

```
map archivo = "datos.txt";

// Escribir en archivo
barrel.write(archivo, "Nuevo dato");

// Leer del archivo
book_and_quill contenido = barrel.read(archivo);
```

Operaciones de números flotantes (Cauldron) Como el caldero que guarda líquidos en cantidades precisas, las operaciones con números flotantes manejan números decimales con exactitud.

Ejemplo:

```
gold_nugget salud = 20.5;
gold_nugget daño = 5.7;

// Operaciones con flotantes usando un caldero
gold_nugget resultado = cauldron(salud - daño);
resultado = cauldron(salud * 0.5);
```

Operaciones de comparación solicitadas (Lectern) Así como el atril compara libros para encontrar el correcto, las operaciones de comparación evalúan si los valores son iguales, mayores o menores entre sí.

Ejemplo:

```
emerald nivel = 10;
emerald limite = 20;

redstone_torch resultado;
resultado = nivel < limite;   // menor que
resultado = nivel == limite;  // igual a
resultado = nivel >= limite;  // mayor o igual
```

Operación de size of (Experience Bar) Como la barra de experiencia mide constantemente cuánta experiencia tiene el jugador, este elemento mide el tamaño de estructuras de datos o longitud de cadenas. Es una forma natural de obtener una medida precisa de cualquier contenedor o secuencia.

Ejemplo:

```
    bundle[3] items = {1, 2, 3};
emerald cantidad = experience_bar(items);  // Obtiene 3
```

Sistema de coherción de tipos (SmithingTable) Gracias a la mesa de herrería se pueden alterar las propiedades de las armaduras, como sería el caso de la coherción de tipos que también influye en la conversión de datos.

Ejemplo:

```
// Conversión de tipos usando la mesa de herrería
emerald entero = diamond(42);
gold_nugget decimal = smithing_table(entero);  // Convierte de entero a flot

gold_nugget pi = splash_potion(3.14159);
emerald redondeado = smithing_table(pi);  // Convierte de flotante a entero
```

0.2.8. Estructuras de Control

Manejo de Bloques de más de una instrucción (CommandBlock) Como el bloque de comandos que ejecuta múltiples acciones en un solo pulso, el manejo de bloques agrupa varias instrucciones para que se ejecuten juntas.

Ejemplo:

```
command_block {
    emerald nivel = diamond(1);
    gold_nugget salud = splash_potion(20.0);
    book_and_quill nombre = sign("Steve");
}
```

Instrucción while (Repeater) Al igual que el repetidor que continúa enviando señales mientras hay energía, el while repite un bloque de código mientras una condición sea verdadera.

Ejemplo:

```
emerald energia = diamond(10);

repeater (energia > 0) {
    // Acciones a repetir
    energia = sword(energia - 1);
}
```

Instrucción if-then-else (Target Block) Como el bloque objetivo que evalúa si se dio en el centro o no, el if-then-else permite ejecutar diferentes bloques de código según una condición. Ejemplo:

```
emerald nivel = diamond(5);

target (nivel >= 10) hit {
    // Si dio en el objetivo (condición verdadera)
    observer(jugador.subir_nivel);
} miss {
    // Si no dio en el objetivo (condición falsa)
    observer(jugador.mantener_nivel);
}
```

Instrucción switch (Jukebox) Como la caja tocadiscos que tiene varias opciones para seleccionar canciones, se puede comparar con la instruccion switch que ejecuta un bloque de codigo segun cierta condicion.

Ejemplo:

```
emerald disco = diamond(2);

jukebox (disco) {
    disc 1: {
        observer(sonido.reproducir_cancion_1);
    }
    disc 2: {
        observer(sonido.reproducir_cancion_2);
    }
    disc 3: {
        observer(sonido.reproducir_cancion_3);
    }
    default: {
        observer(sonido.sin_disco);
    }
}
```

Instrucción Repeat-until (Spawner) Como el generador de monstruos que continúa spawneando criaturas hasta que se cumpla una condición de parada, la instrucción repeat-until ejecuta un bloque de código repetidamente hasta que se cumpla una condición específica.

Ejemplo:

```
emerald mobs = diamond(5);
emerald radio = diamond(10);

spawner {
    observer(zona.generar_mob);
    mobs = sword(mobs - 1);
} exhausted (mobs == 0)
```

Instrucción For (NoteBlock) Como el bucle for los NoteBlocks o cajas musicales, recorren nota por nota hasta terminar la cancion, similar al ciclo for que se ejecuta linea por linea de manera ordenada hasta que acabe su condicional.

Ejemplo:

```
bundle[4] notas = minecart[1, 2, 3, 4];

note_block (emerald i = diamond(0); i < 4; i++) {
    observer(musica.reproducir_nota[i]);
}
```

Instrucción With (Painting) Como la instruccion with es capaz de definir una serie de instrucciones que se pueden mantener para un segmento de codigo, se hizo la analogia con los cuadros, gracias a esto un cuadro puede mantener una forma concreta para decorar una habitacion de forma especial.

Ejemplo:

```
structure Cuadro {
    emerald ancho;
    emerald alto;
    book_and_quill estilo;
}

painting (Cuadro) {
    observer(cuadro.ancho) = diamond(64);
    observer(cuadro.alto) = diamond(32);
    observer(cuadro.estilo) = sign("paisaje");
}
```

Instrucción break (Piston) Este bloque es capaz de poder interrumpir las acciones en ejecucion, como en el juego el bloque de piston es capaz de interrumpir circuitos de redstone.

Ejemplo:

```
repeater (lever_on) {
    emerald energia = diamond(redstone_dust.potencia);

    target (energia < 5) hit {
        piston;  // Interrumpe el ciclo
    }
    observer(maquina.trabajar);
}
```

Instrucción continue (SlimeBlock) Los Slime Blocks son bloques especiales que nos permiten saltar y obtener un efecto rebote para impulsarnos, gracias a esto se hizo la analogia con la instruccion continue.

Ejemplo:

```
note_block (emerald i = diamond(0); i < 10; i++) {
    target (i == 5) hit {
        slime_block;  // Salta a la siguiente iteración
    }
    observer(proceso.ejecutar);
}
```

Instrucción Halt (EndPortal) Como la instruccion Halt es encargada de detener un programa, se tiene el End Portal, que nos lleva a la parte final del juego. Una manera simbolica de decir que ya esta terminado, no obstante, se hace referencia que esta instruccion pone al CPU en un estado inactivo, pero no lo termina por completo.

Ejemplo:

```
spawn {
    // Código principal
    emerald nivel = diamond(100);

    target (nivel > 50) hit {
        // Si el nivel es suficiente, terminamos el programa
        end_portal;  // Detiene la ejecución del programa
    } miss {
        observer(jugador.entrenar);
    }
}
```

0.2.9. Funciones y Procedimientos

Encabezado de funciones (EnchantmentTable) La mesa de encantamientos es capaz de dotar simples objetos en objetos poderosos de alto valor, lo mismo que pueden hacer las funciones que al llegarle una variable pueden devolver otras cosas más complejas.

Ejemplo:

```
enchantment_table emerald calcular_daño(emerald nivel,
gold_nugget multiplicador) {
    // Función que calcula el daño basado en nivel
    // y multiplicador

    gold_nugget daño_base = splash_potion(nivel * multiplicador);
    emerald daño_final = dispenser(daño_base);
    totem_undying(daño_final);
}
```

Encabezado de procedimientos (Grindstone) Este bloque nos da la facilidad de poder hacer varios tipos de manipulaciones como lo puede ser reparar objetos y quitar encantamientos, esto se puede relacionar a las funcionalidades que se pueden definir en los encabezados de procedimientos.

Ejemplo:

```
grindstone reparar_equipo(structure Equipamiento equipo) {
    // Procedimiento para reparar equipo
    observer(equipo.durabilidad) = diamond(100);
    observer(equipo.nivel) = diamond(1);
}
```

Manejo de parámetros formales (Tripwire Hook) Como los parametros formales, sirven para poder detectar los parametros reales en caso de ser enviados, en el juego los ganchos de alambre trampa sirven como sensores que normalmente son utilizados para detectar el movimiento de un mob.

Ejemplo:

```
grindstone intercambiar_items(tripwire_hook emerald item1,
tripwire_hook emerald item2) {
    // Los parámetros formales item1 e item2 detectarán
    // los valores pasados
    emerald temp = item1;
    item1 = item2;
```

```
    item2 = temp;
}
```

Manejo de parámetros reales (EnderPearl) Como la ender pearl ayuda a poder teletransportarse dentro del juego se hace referencia al manejo de parametros reales los cuales son las expresiones que se usan para el envio y manejo de datos.

Ejemplo:

```
// Declaración de variables
emerald espada = diamond(10);
emerald escudo = diamond(5);

// Llamada a la función con parámetros reales
ender_pearl intercambiar_items(espada, escudo);
// Envía los valores mediante teletransporte
```

Instrucción return (TotemUndying) Este mob tiene ciertas habilidades como poder devolver a la vida al jugador, lo que es idea para una analogia como el return, tambien como la funcion de return solo se cumple si hay ciertas condiciones. En el juego la unica manera de que el mob nos devuelva la vida es si el jugador lo esta sosteniendo en la mano.

Ejemplo:

```
enchantment_table emerald calcular_experiencia(emerald nivel) {
    target (nivel <= 0) hit {
        totem_undying(diamond(0));  // Retorna 0 si el nivel es inválido
    }

    emerald exp = sword(nivel * 2);
    totem_undying(exp);  // Retorna la experiencia calculada
}
```

0.2.10. Funciones especiales

Funciones especiales con una funcionalidad definida que pueden ser de ayuda al programador.

Funcion distancia (Coords) segun el tipo de dato Compass que almacena el numero de latitud y longitud y un lugar se puede calcular la distancia entre dos puntos, esto se hace mediante la funcion coords. Se deben de pasar

como parametros dos valores de tipo Compass. a continuacion se muestra un ejemplo:

```
chest {
    egg compass distancia = Coords(distancia1, distancia2);
}
```

Sumatoria (Inventory) La funcion inventory tiene la funcionalidad de la sumatoria, lo que hace es sumar todos los numeros de un rango dado, se deben de dar dos numeros enteros que denotaran desde donde comenzara a sumar hasta donde terminara, se relaciona con el inventario ya que se pueden hacer calculos con la cantidad de stacks que tenemos. A continuacion se presenta un ejemplo:

```
chest {
    emerald sumatoria = inventory(num1, num2);
}
```

Espejo de String (Rollercoaster) La funcion Rollercoaster tiene la funcionalidad de darle vuelta a un string, lo que hace es intercambiar todos los caracteres del string se asigna el nombre de rollercoaster ya que da la sensacion de vueltas, referencia a cuando se le da vuelta a una palabra. A continuacion se muestra un ejemplo

```
chest {
    sign palabraVolteada = Rollercoaster(palabra);
}
```

0.2.11. Entrada/Salida y Elementos Auxiliares

Manejo de la entrada estándar (Villager Request) En Minecraft, los aldeanos pueden solicitar objetos específicos cuando realizamos un intercambio, similar a cómo un programa solicita datos al usuario. Cuando un aldeano muestra lo que necesita, está "solicitando una entrada"del jugador.

```
book_and_quill nombre;
villager_request(nombre); // Solicita y lee un valor del usuario
```

Manejo de la salida estándar (Villager Offer) Como respuesta a nuestras interacciones, los aldeanos nos muestran lo que pueden darnos a cambio. Esto es similar a cómo un programa muestra información al usuario como resultado de una operación.

```
villager_offer(nombre); // Muestra el valor en pantalla
```

Terminador o separador de instrucciones - Instrucción nula (Fence Gate) Es capaz de poder regular el paso de las instrucciones como la puerta de las vallas, gracias a esto se tiene un control de las instrucciones, semejante al separador de instrucciones.

Ejemplo:

```
emerald nivel = 1;
villager_offer(sign("Nivel actual"));
villager_offer(nivel);
```

Todo programa se debe cerrar con un (The End) Se cierra con The End ya que representa el final definitivo del juego en Minecraft, después de derrotar al Ender Dragon. Es una analogía natural para indicar la terminación completa de un programa.

Ejemplo:

```
spawn {
    villager_offer("Cálculos completados");
    the_end
}
```

Comentario de Bloque (GlowInkSac) Es capaz de agregar notas visibles, por lo que es una excelente opcion para resaltar bloques de comentarios que pueden ser extensos o tener documentacion importante.

Ejemplo:

```
/*
    Este programa calcula el nivel del jugador
    y muestra el resultado en pantalla.
    Autor: Steve
    Fecha: 17/02/2024
*/
```

Comentario de Línea (Feather) Los comentarios de linea se deben de relacionar con las plumas ya que deben de ser cortos o ligeros, esto porque solo caben en una linea. Ademas no tienen peso a la hora de ejecucion.

Ejemplo:

```
emerald nivel = 1;  // Inicializa el nivel del jugador
villager_offer(nivel);      // Muestra el nivel actual
```

0.3. Conjunto de pruebas

Se presenta el conjunto de pruebas que sirven para ejemplificar las características del lenguaje EnderLang. Cada archivo ha sido diseñado para demostrar diferentes elementos del lenguaje, desde estructuras básicas hasta características más conplejas.

0.3.1. p01_hola_mundo.txt

Prueba la estructura básica del programa, el punto de entrada (spawn), la salida estándar (villager_offer) y los literales de tipo string (sign).

0.3.2. p02_variables.txt

Demuestra la declaración de variables (item_frame), los tipos básicos (emerald, book_and_quill, redstone_torch, gold_nugget, book) y los literales para cada tipo (diamond, sign, lever_on, splash_potion, name_tag).

0.3.3. p03_constantes.txt

Prueba la declaración de constantes (beacon), constantes de diferentes tipos y el acceso a constantes.

0.3.4. p04_aritmetica.txt

Ejemplifica las operaciones aritméticas con enteros (sword), operaciones aritméticas con flotantes (cauldron) y asignaciones.

0.3.5. p05_condicionales.txt

Demuestra la estructura condicional if-then-else (target-hit-miss), la estructura switch-case (jukebox-disc-default) y el uso de break (piston).

0.3.6. p06_bucles.txt

Prueba los bucles while (repeater), do-while (spawner-exhausted), for (note_block) y las instrucciones break (piston) y continue (slime_block).

0.3.7. p07_funciones.txt

Demuestra la declaración de funciones (enchantment_table), la declaración de procedimientos (grindstone), los parámetros formales (tripwire_hook), el retorno de valores (totem_undying) y la recursividad.

0.3.8. p08_estructuras.txt

Prueba la declaración de estructuras (structure), arreglos (bundle), el acceso a elementos de arreglo (hopper) y el acceso a campos de estructura (observer).

0.3.9. p09_logica.txt

Ejemplifica las operaciones de comparación y las operaciones lógicas (redstone_dust).

0.3.10. p10_strings.txt

Demuestra la concatenación de strings (fishing_rod), el acceso a caracteres en strings (comparator), la modificación de caracteres y la medición de tamaño de string (experience_bar).

0.3.11. p11_entrada_salida.txt

Prueba la entrada estándar (villager_request), la salida estándar (villager_offer) y la interacción con el usuario.

0.3.12. p12_tipos.txt

Demuestra la conversión entre tipos (anvil, smithing_table) y la conversión implícita (dispenser).

0.3.13. p13_completo.txt

Presenta un programa completo con todas las secciones y el uso combinado de múltiples características del lenguaje.

0.3.14. p14_conjuntos.txt

Ejemplifica las operaciones específicas con conjuntos: unión, intersección y diferencia (campfire).

0.3.15. p15_archivos.txt

Demuestra las operaciones con archivos: escritura y lectura (barrel).

0.3.16. p16_caracteres.txt

Prueba las funciones específicas para caracteres: conversión a mayúsculas, obtención de código ASCII y verificación de tipo (book_shelf).

0.3.17. p17_with.txt

Ejemplifica el uso de la instrucción with (painting) para manipular estructuras.

0.3.18. p18_errores.txt

Demuestra el manejo básico de errores con estructuras condicionales.

0.3.19. p19_recursividad.txt

Prueba las funciones recursivas, la programación funcional y la manipulación de arreglos.

0.3.20. p20_conversion_tipos.txt

Ejemplifica el sistema completo de conversión de tipos: explícita, implícita y entre diferentes tipos básicos.

0.3.21. p21_alcance.txt

Demuestra el alcance (scope) de variables en bloques anidados y el manejo de variables locales.

El conjunto de pruebas cubre varios de los elementos definidos en la gramática BNF del lenguaje, desde la estructura básica hasta características más avanzadas como manejo de memoria, alcance de variables, y operaciones específicas para cada tipo de dato.