TEC | Tecnológico de Costa Rica

Instituto Tecnológico de Costa Rica

Escuela de Computación

# Compiladores e Intérpretes, IC5701

**Estudiantes:**
Samir Cabrera, 2022161229
Luis Urbina, 2023156802

Primer semestre del año 2025

# Índice general

## 0.1. Concurso de Nombres

**EnderLang**
**Logo:**

**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. 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
    };
}
```

### 0.2.5. 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
```

### 0.2.6. 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
```

### 0.2.7. 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);
    }
}
```

### 0.2.8. 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
}
```

### 0.2.9. 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. Gramática BNF

Antes de comenzar, es importante entender los símbolos que se usaran:

- `::=` significa "se define como"
- | significa "o" (alternativa)
- ⟨ ⟩ encierran nombres de elementos
- ‘‘’’ encierran texto literal que aparecerá en el programa
- $\varepsilon$ significa "vacío" (que puede no haber nada)

### 0.3.1. Elementos Sintácticos Básicos

```
<separador> ::= ";"

<identificador> ::= <letra> <resto_identificador>

<resto_identificador> ::= <letra> <resto_identificador>
                        | <digito> <resto_identificador>
                        | <caracter_especial_permitido> <resto_identificador>
                        |

<letra> ::= "a" | "b" | "c" | "d" | "e" | "f" | "g" | "h" | "i" | "j"
| "k" | "l" | "m" | "n" | "o" | "p" | "q" | "r" | "s" | "t" | "u" | "v"
| "w" | "x" | "y" | "z"| "A" | "B" | "C" | "D" | "E" | "F" | "G" | "H"
| "I" | "J" | "K" | "L" | "M" | "N" | "O" | "P" | "Q" | "R" | "S" | "T"
| "U" | "V" | "W" | "X" | "Y" | "Z"

<digito> ::= "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"

<caracter_especial_permitido> ::= "_"
```

```
<espacio_blanco> ::= " " | "\t" | "\n" | "\r" | "\f"

<secuencia_espacios> ::= <espacio_blanco> <secuencia_espacios>
                       | <espacio_blanco>

<fin_de_linea> ::= "\n" | "\r\n" | "\r"

<cualquier_caracter> ::= <letra> | <digito> | <simbolo> | <espacio_blanco>

<simbolo> ::= "!" | "@" | "#" | "$" | "%" | "^" | "&" | "*" | "(" | ")"
| "-" | "+" | "=" | "{" | "}" | "[" | "]" | ":" | ";" | "\"" | "'" | "<"
| ">" | "," | "." | "/" | "?" | "|" | "\\" | "~" | "`"

<palabra_reservada> ::= "spawn" | "the_end" | "bedrock" | "shulker_box"
| "chest" | "crafting_table" | "redstone_circuit" | "beacon" | "anvil"
| "item_frame" | "emerald" | "book" | "book_and_quill" | "redstone_torch"
| "banner_pattern" | "map" | "gold_nugget" | "bundle" | "structure"
| "lever_on" | "lever_off" | "firework_star" | "book_item" | "splash_potion"
| "diamond" | "name_tag" | "sign" | "minecart" | "armor_stand"
| "command_block" | "repeater" | "target" | "hit" | "miss"| "jukebox"
| "disc" | "default" | "spawner" | "exhausted" | "note_block" | "painting"
| "piston" | "slime_block" | "end_portal" | "enchantment_table"
| "grindstone" | "tripwire_hook" | "ender_pearl" | "totem_undying"
| "villager_request" | "villager_offer" | "hopper" | "comparator"
| "observer" | "cauldron" | "book_shelf" | "redstone_dust"
| "campfire" | "barrel" |  "experience_bar"
| "smithing_table" | "dispenser"
```

**Explicación simple:** Esta sección define los elementos sintácticos básicos del lenguaje:

- El separador de instrucciones es el punto y coma (;)
- Un identificador debe comenzar con una letra y puede contener letras, dígitos o guiones bajos
- Se definen los caracteres permitidos: letras, dígitos y símbolos
- Los espacios en blanco incluyen espacios, tabulaciones y saltos de línea
- Se listan todas las palabras reservadas que no pueden usarse como identificadores

### 0.3.2. Ejemplos de Uso

```
// Ejemplos de identificadores válidos
player1
vida_jugador
inventarioPrincipal
alturaMaxima
x
_temporal
jugadorNPC123
mapa_nivel_1

// Ejemplos de identificadores inválidos
1player       // No puede comenzar con número
mi-variable   // El guión no está permitido
miss          // Es una palabra reservada
gold_nugget         // Es una palabra reservada

// Ejemplos de uso del separador
emerald nivel = 1;
gold_nugget salud = 20.5; villager_offer(salud);

// Ejemplos de uso de espacios (no afectan la sintaxis)
emerald  contador=0;  // Espacios irregulares pero válidos
gold_nugget
    temperatura = 36.6;  // Con salto de línea
```

### 0.3.3. Estructura Básica del Programa

```
<programa> ::= <secciones> <bloque_principal>

<bloque_principal> ::= "spawn" <bloque> "the_end"

<bloque> ::= "{" <lista_instrucciones> "}"
          | "{" "}"

<lista_instrucciones> ::= <instruccion> <lista_instrucciones>
                        | <instruccion>

<instruccion> ::= <instruccion_simple> ";"
                | <instruccion_compuesta>
```

```
                | <instruccion_control>

<secciones> ::= <seccion_bedrock> <resto_secciones>
              | <resto_secciones>

<resto_secciones> ::= <seccion_shulker> <resto_secciones_2>
                    | <resto_secciones_2>

<resto_secciones_2> ::= <seccion_chest> <resto_secciones_3>
                      | <resto_secciones_3>

<resto_secciones_3> ::= <seccion_crafting_table> <resto_secciones_4>
                      | <resto_secciones_4>

<resto_secciones_4> ::= <seccion_redstone>
                      |

<seccion_bedrock> ::= "bedrock" "{" <lista_constantes> "}"
                    | "bedrock" "{" "}"

<seccion_shulker> ::= "shulker_box" "{" <lista_tipos> "}"
                    | "shulker_box" "{" "}"

<seccion_chest> ::= "chest" "{" <lista_variables> "}"
                  | "chest" "{" "}"

<seccion_crafting_table> ::= "crafting_table" "{" <lista_prototipos> "}"
                           | "crafting_table" "{" "}"

<seccion_redstone> ::= "redstone_circuit" "{" <lista_rutinas> "}"
                     | "redstone_circuit" "{" "}"

<lista_prototipos> ::= <prototipo> ";" <lista_prototipos>
                     | <prototipo> ";"
                     |

<prototipo> ::= <declaracion_funcion_prototipo>
              | <declaracion_procedimiento_prototipo>

<declaracion_funcion_prototipo> ::= "enchantment_table" <tipo_retorno>
<identificador> "(" <lista_parametros> ")"
```

```
<declaracion_procedimiento_prototipo> ::= "grindstone" <identificador>
"(" <lista_parametros> ")"

<lista_rutinas> ::= <rutina> <lista_rutinas>
                  | <rutina>
                  |

<rutina> ::= <declaracion_funcion>
           | <declaracion_procedimiento>
```

**Explicación simple:** La estructura básica de un programa en EnderLang consiste en:

- Secciones opcionales que deben seguir un orden específico: bedrock (constantes), shulker_box (tipos), chest (variables), crafting_table (prototipos) y redstone_circuit (rutinas)
- Un bloque principal obligatorio que comienza con "*spawn*" y termina con "the_end"
- Cada sección contiene elementos específicos como constantes, tipos, variables, prototipos y rutinas
- Los prototipos son declaraciones de funciones o procedimientos sin cuerpo
- Las rutinas son implementaciones completas de funciones o procedimientos
- Cada sección puede estar vacía

### 0.3.4. Ejemplo de Programa Básico

```
// Ejemplo de programa completo con todas las secciones

// Sección de constantes (opcional)
bedrock {
    beacon MAX_NIVEL = 100;
    beacon MIN_NIVEL = 1;
    beacon VERSION = "v1.0";
}
```

```
// Sección de tipos (opcional)
shulker_box {
    anvil gold_nugget -> emerald;
    anvil emerald -> book_and_quill;
}

// Sección de variables (opcional)
chest {
    item_frame vida = 20;
    item_frame nombre = "Steve";
    item_frame nivel;
}

// Sección de prototipos (opcional)
crafting_table {
    enchantment_table emerald calcular_daño(tripwire_hook emerald nivel,
    tripwire_hook gold_nugget multiplicador);
    grindstone mostrar_estado(tripwire_hook book_and_quill nombre,
    tripwire_hook emerald vida);
}

// Sección de rutinas (opcional)
redstone_circuit {
    // Implementación de la función
    enchantment_table emerald calcular_daño(
        tripwire_hook emerald nivel,
        tripwire_hook gold_nugget multiplicador
    ) {
        gold_nugget daño_base = splash_potion(nivel * multiplicador);
        emerald daño_final = dispenser(daño_base);
        totem_undying(daño_final);
    }

    // Implementación del procedimiento
    grindstone mostrar_estado(
        tripwire_hook book_and_quill nombre,
        tripwire_hook emerald vida
    ) {
        villager_offer(nombre);
        villager_offer(vida);
    }
```

```
}

// Bloque principal (obligatorio)
spawn {
    // Solicitar nombre al usuario
    villager_request(nombre);

    // Calcular nivel basado en la vida
    nivel = calcular_daño(vida, splash_potion(0.5));

    // Mostrar estado del jugador
    ender_pearl mostrar_estado(nombre, nivel);

    // Mensaje de despedida
    villager_offer(sign("Adiós, ") + nombre + sign("!"));
} the_end

// Ejemplo de programa mínimo (sólo con el bloque principal)
spawn {
    villager_offer(sign("Hola Mundo"));
} the_end
```

### 0.3.5. Declaraciones

#### Declaración de Constantes

```
<lista_constantes> ::= <declaracion_constante> ";" <lista_constantes>
                     | <declaracion_constante> ";"
                     |

<declaracion_constante> ::= "beacon" <identificador> "=" <expresion_constante>

<expresion_constante> ::= <literal_entero>
                        | <literal_flotante>
                        | <literal_caracter>
                        | <literal_string>
                        | <literal_booleano>
                        | <literal_conjunto>
                        | <expresion_constante_aritmetica>
                        | <identificador_constante>
```

```
<expresion_constante_aritmetica> ::=
<expresion_constante> "+" <expresion_constante>
| <expresion_constante> "-" <expresion_constante>
| <expresion_constante> "*" <expresion_constante>
| <expresion_constante> "/" <expresion_constante>
| "(" <expresion_constante_aritmetica> ")"

<identificador_constante> ::= <identificador>
```

**Explicación simple:** Las constantes se declaran con la palabra clave "beacon"seguida de un nombre y un valor. El valor puede ser cualquier literal o una expresión constante (que puede incluir operaciones aritméticas con otras constantes). Una vez definida, una constante no puede cambiar su valor.

### Declaración de Tipos

```
<declaracion_tipo> ::= "anvil" <tipo_origen> "->" <tipo_destino>
| "anvil" <tipo_origen> "->" <tipo_destino> <modo_conversion>

<modo_conversion> ::= "truncate" | "round" | "safe"

<tipo_origen> ::= <tipo_dato>
<tipo_destino> ::= <tipo_dato>

<tipo_lista> ::= <declaracion_tipo> ";" <tipo_lista>
               | <declaracion_tipo> ";"
```

**Explicación simple:** Los tipos de conversión se definen usando .ªnvil"para indicar una transformación de un tipo a otro. Se puede especificar el modo de conversión para controlar cómo se manejan posibles pérdidas de datos durante la conversión.

### Declaración de Variables

```
<lista_variables> ::= <declaracion_variable> ";" <lista_variables>
                    | <declaracion_variable> ";"
                    |

<declaracion_variable> ::= "item_frame" <identificador> "=" <expresion>
| "item_frame" <identificador>
| "item_frame" <lista_identificadores>
| "item_frame" <tipo_dato> <identificador> "=" <expresion>
```

```
| "item_frame" <tipo_dato> <identificador>

<lista_identificadores> ::= <identificador> "," <lista_identificadores>
                          | <identificador>
```

**Explicación simple:** Las variables se declaran con ïtem_frame"seguido por un identificador y opcionalmente un valor inicial. También se pueden declarar múltiples variables del mismo tipo en una sola línea usando una lista de identificadores. De manera opcional, se puede especificar explícitamente el tipo de dato.

### 0.3.6. Ejemplos de Declaraciones

```
// Declaración de constantes
bedrock {
    beacon MAX_SALUD = 100;
    beacon MIN_NIVEL = 1;
    beacon PI = 3.14159;
    beacon NOMBRE_JUEGO = "EnderLang";
    beacon GRAVEDAD = -9.8;
    beacon MENSAJE_BIENVENIDA = "Bienvenido a " + NOMBRE_JUEGO;
}

// Declaración de tipos y conversiones
shulker_box {
    anvil gold_nugget -> emerald truncate;
    // Trunca el decimal al convertir a entero

    anvil emerald -> book_and_quill;

    anvil book -> emerald safe;
    // Conversión segura de caracter a código ASCII
}

// Declaración de variables
chest {
    // Declaración simple
    item_frame vida = 20;
    item_frame nombre = "Steve";
    item_frame nivel;
```

```
    // Declaración de múltiples variables
    item_frame x, y, z;

    // Declaración con tipo explícito
    item_frame emerald contador = 0;
    item_frame gold_nugget temperatura = 36.5;

    // Declaración con expresión compleja
    item_frame posicion = x * 100 + y * 10 + z;
}
```

### 0.3.7. Tipos de Datos y Literales

```
// Tipos de Datos Básicos
<tipo_dato> ::= <tipo_basico> | <tipo_compuesto>

<tipo_basico> ::= "emerald"           /* entero */
                | "book"              /* caracter */
                | "book_and_quill"    /* string */
                | "redstone_torch"    /* booleano */
                | "banner_pattern"    /* conjunto */
                | "map"               /* archivo */
                | "gold_nugget"       /* flotante */

// Tipos de Datos Compuestos
<tipo_compuesto> ::= <tipo_arreglo> | <tipo_registro>

<tipo_arreglo> ::= "bundle" "[" <expresion_entera> "]" <tipo_dato>

<tipo_registro> ::= "structure" <identificador> "{" <lista_campos> "}"
                  | "structure" <identificador>

<lista_campos> ::= <declaracion_campo> ";" <lista_campos>
                | <declaracion_campo> ";"

<declaracion_campo> ::= <tipo_dato> <identificador>

// Literales
<literal> ::= <literal_entero>
            | <literal_caracter>
            | <literal_string>
```

```
            | <literal_booleano>
            | <literal_conjunto>
            | <literal_archivo>
            | <literal_flotante>
            | <literal_arreglo>
            | <literal_registro>

// Literales Enteros
<literal_entero> ::= <numero>
                   | "diamond" "(" <numero> ")"
                   | "diamond" "(" <expresion_entera> ")"
                   | "-" <literal_entero>

<numero> ::= <digito> <resto_numero>
<resto_numero> ::= <digito> <resto_numero> |

// Literales Caracteres
<literal_caracter> ::= "name_tag" "(" "'" <caracter> "'" ")"
| "name_tag" "(" <expresion_caracter> ")"

<caracter> ::= <letra> | <digito> | <simbolo>

// Literales String
<literal_string> ::="sign" "(" "\"" <texto> "\"" ")"
| "sign" "(" <expresion_string> ")"

<texto> ::= <caracter> <texto> |

// Literales Booleanos
<literal_booleano> ::= "lever_on"   /* true */
| "lever_off"  /* false */

// Literales Conjunto
<literal_conjunto> ::=
"firework_star" "{" <lista_elementos> "}"
| "firework_star" "(" <expresion_conjunto> ")"

<lista_elementos> ::= <expresion> "," <lista_elementos>
                    | <expresion>
                    |
```

```
// Literales Archivo
<literal_archivo> ::= "book_item" "(" <literal_string> ")"
| "book_item" "(" <expresion_string> ")"

// Literales Flotantes
<literal_flotante> ::= "splash_potion" "(" <numero_decimal> ")"
| "splash_potion" "(" <expresion_flotante> ")"

<numero_decimal> ::= <numero> "." <numero>
                   | <numero> "."
                   | "." <numero>

// Literales Arreglo
<literal_arreglo> ::= "{" <lista_elementos> "}"
                    | "minecart" "[" <lista_elementos> "]"
                    | "minecart" "(" <expresion_arreglo> ")"

// Literales Registro
<literal_registro> ::= "{" <lista_campos_valor> "}"
                     | "armor_stand" "{" <lista_campos_valor> "}"
                     | "armor_stand" "(" <expresion_registro> ")"

<lista_campos_valor> ::= <campo_valor> "," <lista_campos_valor>
                       | <campo_valor>
                       |

<campo_valor> ::= <identificador> ":" <expresion>
```

**Explicación simple:** Esta sección define todos los tipos de datos disponibles en el lenguaje y cómo escribir sus valores literales:

- Tipos básicos: emerald (entero), book (caracter), book_and_quill (string), redstone_torch (booleano), banner_pattern (conjunto), map (archivo), gold_nugget (flotante)
- Tipos compuestos: bundle (arreglos), structure (registros)
- Literales para cada tipo con sus formas especiales de Minecraft:
  - Enteros: con diamond()
  - Caracteres: con name_tag()
  - Strings: con sign()

- Booleanos: con lever_on y lever_off
- Conjuntos: con firework_star
- Archivos: con book_item
- Flotantes: con splash_potion
- Arreglos: con minecart
- Registros: con armor_stand

### 0.3.8. Ejemplos de Uso de Tipos y Literales

```
// Ejemplos de tipos básicos
emerald contador;
book letra;
book_and_quill nombre;
redstone_torch encendido;
banner_pattern materiales;
map archivo;
gold_nugget precio;

// Ejemplos de tipos compuestos
bundle[10] inventario;
structure Jugador {
    book_and_quill nombre;
    emerald nivel;
    gold_nugget salud;
};
structure Jugador jugador;

// Ejemplos de literales enteros
emerald nivel = 10;
emerald negativo = -5;
emerald creado = diamond(5 * 2);  // 10, usando función diamond

// Ejemplos de literales caracteres
book simbolo = name_tag('$');
book calculado = name_tag(65);  // 'A', usando código ASCII

// Ejemplos de literales string
book_and_quill cartel = sign("Bienvenido");
book_and_quill combinado = sign(nombre + " está jugando");
```

```
// Ejemplos de literales booleanos
redstone_torch encendido = lever_on;
redstone_torch apagado = lever_off;

// Ejemplos de literales conjunto
banner_pattern minerales = firework_star{hierro, oro, diamante};

// Ejemplos de literales archivo
map config = book_item("configuracion.dat");

// Ejemplos de literales flotantes
gold_nugget pi = 3.14159;
gold_nugget negativo = -2.5;
gold_nugget pocion = splash_potion(2.5);

// Ejemplos de literales arreglo
bundle[4] items = minecart[1, 2, 3, 4];

// Ejemplos de literales registro
structure Equipo equipo = {nombre: "Diamante", nivel: 3};
structure Herramienta herramienta = armor_stand{
    tipo: "pico",
    material: "hierro",
    durabilidad: 100
};
```

### 0.3.9. Expresiones y Operadores

```
// Expresiones generales
<expresion> ::= <literal>
              | <identificador>
              | <expresion_operacion>
              | <expresion_acceso>
              | <llamada_funcion>
              | "(" <expresion> ")"

<expresion_operacion> ::= <expresion_aritmetica>
                        | <expresion_logica>
                        | <expresion_comparacion>
                        | <expresion_string>
```

```
                        | <expresion_conjunto>
                        | <expresion_incremento>

// Operaciones aritméticas con enteros (sword)
<expresion_aritmetica_enteros> ::=
"sword" "(" <expresion> "+" <expresion> ")"
| "sword" "(" <expresion> "-" <expresion> ")"
| "sword" "(" <expresion> "*" <expresion> ")"
| "sword" "(" <expresion> "/" <expresion> ")"
| "sword" "(" <expresion> "%" <expresion> ")"

// Operaciones aritméticas con flotantes (cauldron)
<expresion_aritmetica_flotantes> ::=
<expresion> "cauldron(" "+" <expresion> ")"
| <expresion> "cauldron(" "-" <expresion> ")"
| <expresion> "cauldron(" "*" <expresion> ")"
| <expresion> "cauldron(" "/" <expresion> ")"

// Incremento y decremento
<expresion_incremento> ::= <identificador> "++"
                         | <identificador> "--"
                         | "++" <identificador>
                         | "--" <identificador>

// Operaciones sobre caracteres (book_shelf)
<expresion_caracter> ::=
<expresion> "book_shelf(" <operacion_caracter> ")"

<operacion_caracter> ::=| "toCode" "(" ")"
                        | "isLetter" "(" ")"

// Operaciones lógicas (redstone_dust)
<expresion_logica> ::=
<expresion> "redstone_dust(" "&&" <expresion> ")"
| <expresion> "redstone_dust(" "||" <expresion> ")"
| "redstone_dust(" "!" <expresion> ")"

// Operaciones de strings (fishing_rod)
<expresion_string> ::= <expresion> "fishing_rod(" "+" <expresion> ")"

// Operaciones de conjuntos (campfire)
```

```
<expresion_conjunto> ::=
<expresion> "campfire(" "+" <expresion> ")"  // unión
| <expresion> "campfire(" "*" <expresion> ")"  // intersección
| <expresion> "campfire(" "-" <expresion> ")"  // diferencia

// Operaciones de archivos (barrel)
<operacion_archivo> ::=
"barrel" "." "write" "(" <expresion> "," <expresion> ")"
| "barrel" "." "read" "(" <expresion> ")"

// Operaciones de comparación
<expresion_comparacion> ::= <expresion> "==" <expresion>
                          | <expresion> "!=" <expresion>
                          | <expresion> "<" <expresion>
                          | <expresion> ">" <expresion>
                          | <expresion> "<=" <expresion>
                          | <expresion> ">=" <expresion>

// Operación de tamaño (experience_bar)
<expresion_tamano> ::= "experience_bar" "(" <expresion> ")"

// Operaciones de acceso
<expresion_acceso> ::= <expresion_acceso_arreglo>
                     | <expresion_acceso_string>
                     | <expresion_acceso_registro>

<expresion_acceso_arreglo> ::=
"hopper" "(" <identificador> "[" <expresion> "]" ")"

<expresion_acceso_string> ::=
"comparator" "(" <identificador> "[" <expresion> "]" ")"

<expresion_acceso_registro> ::=
"observer" "(" <identificador> "." <identificador> ")"

// Coerción de tipos (smithing_table)
<expresion_coercion> ::= "smithing_table" "(" <expresion> ")"

// Asignación
<asignacion> ::= <identificador> "=" <expresion>
               | <expresion_acceso_arreglo> "=" <expresion>
```

```
                | <expresion_acceso_string> "=" <expresion>
                | <expresion_acceso_registro> "=" <expresion>

// Asignación con despacho de tipos (dispenser)
<asignacion_dispenser> ::=
<identificador> "=" "dispenser" "(" <expresion> ")"
```

**Explicación simple:** Esta sección define las operaciones que se pueden realizar en el lenguaje, basándose fielmente en la definición original:

- Operaciones aritméticas con enteros usando sword
- Operaciones aritméticas con flotantes usando cauldron
- Incremento y decremento usando
- Operaciones sobre caracteres usando book_shelf
- Operaciones lógicas usando redstone_dust
- Concatenación de strings usando fishing_rod
- Operaciones de conjuntos usando campfire
- Operaciones de archivos usando barrel
- Comparaciones
- Medición de tamaño usando experience_bar
- Acceso a estructuras de datos usando hopper, comparator y observer
- Coerción de tipos usando smithing_table
- Asignaciones normales y con dispatch (dispenser)

### 0.3.10. Ejemplos de Uso de Expresiones y Operadores

```
// Ejemplos de operaciones aritméticas
emerald resultado = sword(nivel + 5);
emerald producto = sword(precio * cantidad);
gold_nugget suma = cauldron(precio + descuento);

// Ejemplos de incremento/decremento
contador++;
```

```
emerald valor = --indice;

// Ejemplos de operaciones de caracteres
book mayuscula = book_shelf(letra.toUpper());
redstone_torch esLetra = book_shelf(letra.isLetter());

// Ejemplos de operaciones lógicas
redstone_torch resultado = redstone_dust(condicion1 && condicion2);
redstone_torch negacion = redstone_dust(!condicion);

// Ejemplos de operaciones de strings
book_and_quill mensaje = fishing_rod(nombre + " está jugando");

// Ejemplos de operaciones de conjuntos
banner_pattern union = campfire(conjunto1 + conjunto2);

// Ejemplos de operaciones de archivos
barrel.write(archivo, datos);
book_and_quill contenido = barrel.read(archivo);

// Ejemplos de comparaciones
redstone_torch esValido = (edad >= 18);

// Ejemplos de acceso
emerald elemento = hopper(arreglo[0]);
book caracter = comparator(texto[5]);
gold_nugget salud = observer(jugador.vida);

// Ejemplo de operación de tamaño
emerald longitud = experience_bar(texto);

// Ejemplos de coerción de tipos
emerald entero = smithing_table(flotante);

// Ejemplos de asignación
contador = diamond(0);
hopper(array[0]) = diamond(10);

// Ejemplo de asignación con dispatch
gold_nugget decimal = dispenser(entero);
// Conversión automática según tipo
```

### 0.3.11. Estructuras de Control

```
<instruccion_control> ::= <instruccion_if>
                        | <instruccion_switch>
                        | <instruccion_while>
                        | <instruccion_do_while>
                        | <instruccion_for>
                        | <instruccion_with>
                        | <instruccion_break>
                        | <instruccion_continue>
                        | <instruccion_halt>

// Manejo de Bloques
<instruccion_compuesta> ::= "command_block" <bloque>
                          | <bloque>

<bloque> ::= "{" <lista_instrucciones> "}"
           | "{" "}"

<lista_instrucciones> ::= <instruccion> <lista_instrucciones>
                        | <instruccion>

// Instrucción if-then-else
<instruccion_if> ::=
"target" "(" <expresion> ")" "hit" <bloque>
| "target" "(" <expresion> ")" "hit" <bloque> "miss" <bloque>

// Instrucción switch
<instruccion_switch> ::=
"jukebox" "(" <expresion> ")" "{" <lista_casos> "}"

<lista_casos> ::= <caso> <lista_casos>
                | <caso>
                |

<caso> ::= "disc" <literal_entero> ":" <bloque>
         | "default" ":" <bloque>

// Instrucción while
<instruccion_while> ::= "repeater" "(" <expresion> ")" <bloque>
```

```
// Instrucción do-while
<instruccion_do_while> ::=
"spawner" <bloque> "exhausted" "(" <expresion> ")"

// Instrucción for
<instruccion_for> ::=
"note_block" "(" <inicializacion> ";"
<condicion> ";" <incremento> ")" <bloque>

<inicializacion> ::= <tipo_dato> <identificador> "=" <expresion>
                   | <asignacion>
                   |

<condicion> ::= <expresion>
              |

<incremento> ::= <expresion_incremento>
               | <asignacion>
               |

// Instrucción with
<instruccion_with> ::= "painting" "(" <identificador_tipo> ")" <bloque>

<identificador_tipo> ::= <identificador>

// Instrucciones de control de flujo
<instruccion_break> ::= "piston" ";"

<instruccion_continue> ::= "slime_block" ";"

<instruccion_halt> ::= "end_portal" ";"
                     | "end_portal" "(" <expresion> ")" ";"
```

**Explicación simple:** Esta sección define todas las estructuras de control disponibles en el lenguaje:

- **command_block**: Agrupa múltiples instrucciones en un bloque
- **target (hit/miss)**: Estructura condicional if-then-else
- **jukebox**: Estructura de selección múltiple (switch-case)

- **repeater**: Bucle while que se ejecuta mientras una condición sea verdadera
- **spawner**: Bucle do-while que se ejecuta al menos una vez
- **note_block**: Bucle for con inicialización, condición e incremento
- **painting**: Estructura with para acceder a campos de un registro
- **piston**: Interrumpe la ejecución de un bucle (break)
- **slime_block**: Salta a la siguiente iteración de un bucle (continue)
- **end_portal**: Termina la ejecución del programa (halt)

### 0.3.12. Ejemplos de Estructuras de Control

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

// Ejemplo de target (if-then-else)
emerald nivel = diamond(5);

// If simple
target (nivel >= 10) hit {
    villager_offer(sign("Nivel suficiente"));
}

// If-else completo
target (nivel >= 10) hit {
    // Si la condición es verdadera
    villager_offer(sign("Nivel suficiente"));
    observer(jugador.subir_nivel);
} miss {
    // Si la condición es falsa
    villager_offer(sign("Nivel insuficiente"));
    observer(jugador.mantener_nivel);
}
```

```
// Ejemplo de jukebox (switch)
emerald disco = diamond(2);

jukebox (disco) {
    disc 1: {
        villager_offer(sign("Reproduciando canción 1"));
        observer(sonido.reproducir_cancion_1);
        piston;  // break para salir del switch
    }
    disc 2: {
        villager_offer(sign("Reproduciando canción 2"));
        observer(sonido.reproducir_cancion_2);
        piston;
    }
    disc 3: {
        villager_offer(sign("Reproduciando canción 3"));
        observer(sonido.reproducir_cancion_3);
        piston;
    }
    default: {
        villager_offer(sign("No hay disco"));
        observer(sonido.sin_disco);
        piston;
    }
}

// Ejemplo de repeater (while)
emerald energia = diamond(10);

repeater (energia > 0) {
    // Acciones a repetir mientras energia > 0
    villager_offer(sign("Energía restante: ") + energia);
    energia = sword(energia - 1);
}

// Ejemplo de spawner (do-while)
emerald mobs = diamond(5);

spawner {
    // Estas acciones se ejecutan al menos una vez
    villager_offer(sign("Generando mob..."));
```

```
    observer(zona.generar_mob);
    mobs = sword(mobs - 1);
} exhausted (mobs == 0)  // Condición de salida

// Ejemplo de note_block (for)
note_block (emerald i = diamond(0); i < 5; i++) {
    villager_offer(sign("Iteración: ") + i);

    // Ejemplo de slime_block (continue)
    target (i == 2) hit {
        villager_offer(sign("Saltando iteración 2"));
        slime_block;  // Salta a la siguiente iteración
    }

    villager_offer(sign("Procesando iteración: ") + i);
}

// Ejemplo de painting (with)
structure Cuadro {
    emerald ancho;
    emerald alto;
    book_and_quill estilo;
};

structure Cuadro cuadro;

painting (Cuadro) {
    // Dentro del with, se accede directamente a los campos
    observer(cuadro.ancho) = diamond(64);
    observer(cuadro.alto) = diamond(32);
    observer(cuadro.estilo) = sign("paisaje");
}

// Ejemplo de piston (break) en un bucle infinito
repeater (lever_on) {  // Bucle infinito
    emerald energia = observer(redstone_dust.potencia);

    target (energia < 5) hit {
        villager_offer(sign("Energía insuficiente, terminando"));
        piston;  // Interrumpe el bucle
    }
```

```
    observer(maquina.trabajar);
}

// Ejemplo de end_portal (halt)
target (nivel > 50) hit {
    villager_offer(sign("Nivel máximo alcanzado, terminando programa"));
    end_portal;  // Detiene la ejecución del programa
}
```

### 0.3.13. Funciones y Procedimientos

```
// Declaración de Funciones
<declaracion_funcion> ::=
"enchantment_table" <tipo_retorno> <identificador>
"(" <lista_parametros> ")" <bloque_funcion>

<tipo_retorno> ::= <tipo_dato>

<bloque_funcion> ::= "{" <lista_instrucciones_funcion> "}"
                   | "{" "}"

<lista_instrucciones_funcion> ::=
<instruccion_funcion> <lista_instrucciones_funcion>
| <instruccion_funcion>

<instruccion_funcion> ::= <instruccion>
| <instruccion_return>

// Declaración de Procedimientos
<declaracion_procedimiento> ::=
"grindstone" <identificador>
"(" <lista_parametros> ")" <bloque_procedimiento>

<bloque_procedimiento> ::= "{" <lista_instrucciones_procedimiento> "}"
                         | "{" "}"

<lista_instrucciones_procedimiento> ::=
<instruccion> <lista_instrucciones_procedimiento>
| <instruccion>
```

```
// Parámetros
<lista_parametros> ::= <parametro> "," <lista_parametros>
                     | <parametro>
                     |

<parametro> ::= "tripwire_hook" <tipo_dato> <identificador>

// Llamada a Funciones y Procedimientos
<llamada_funcion> ::=
<identificador> "(" <lista_argumentos> ")"

<llamada_procedimiento> ::=
"ender_pearl" <identificador> "(" <lista_argumentos> ")"

<lista_argumentos> ::= <argumento> "," <lista_argumentos>
                     | <argumento>
                     |

<argumento> ::= <expresion>

// Instrucción de Retorno
<instruccion_return> ::= "totem_undying" "(" <expresion> ")" ";"
                       | "totem_undying" ";"

// Entrada/Salida Estándar
<instruccion_entrada> ::=
"villager_request" "(" <identificador> ")" ";"

<instruccion_salida> ::=
"villager_offer" "(" <expresion> ")" ";"
```

**Explicación simple:** Esta sección define cómo declarar y usar funciones y procedimientos:

- Funciones usando .enchantment_table"(retornan valor)
- Procedimientos usando "grindstone"(no retornan valor)
- Parámetros formales usando "tripwire_hook"
- Llamadas a procedimientos usando .ender_pearl"

- Retorno de valores usando "totem_undying"
- Entrada/salida usando "villager_requestz "villager_offer"

### 0.3.14. Ejemplos de Uso de Funciones y Procedimientos

```
// Ejemplo de función
enchantment_table emerald calcular_daño(
    tripwire_hook emerald nivel,
    tripwire_hook gold_nugget multiplicador
) {
    gold_nugget daño_base = splash_potion(nivel * multiplicador);
    emerald daño_final = dispenser(daño_base);
    totem_undying(daño_final);
}

// Ejemplo de procedimiento
grindstone reparar_equipo(
    tripwire_hook structure Equipamiento equipo
) {
    observer(equipo.durabilidad) = diamond(100);
    observer(equipo.nivel) = diamond(1);
}

// Ejemplo de llamada a función
emerald daño = calcular_daño(nivel, multiplicador);

// Ejemplo de llamada a procedimiento
ender_pearl reparar_equipo(equipo_actual);

// Ejemplos de entrada/salida
book_and_quill nombre;
villager_request(nombre);  // Lee entrada
villager_offer(nombre);    // Muestra salida

// Ejemplo de función con retorno condicional
enchantment_table emerald verificar_nivel(
    tripwire_hook emerald nivel
) {
    target (nivel <= 0) hit {
```

```
        totem_undying(diamond(0));  // Retorno temprano
    }

    emerald nivel_final = sword(nivel * 2);
    totem_undying(nivel_final);  // Retorno normal
}
```

### 0.3.15. Manejo de Memoria y Alcance

```
// Declaración de Alcance
<alcance> ::= "{" <lista_declaraciones> "}"
<lista_declaraciones> ::= <declaracion> <lista_declaraciones>
                        | <declaracion>
                        |
<declaracion> ::= <declaracion_variable>
                | <declaracion_funcion>
                | <declaracion_procedimiento>
// Coerción de Tipos
<coercion_tipo> ::=
"(" <tipo_dato> ")" <expresion>
| "smithing_table" "(" <expresion> ")"
| "smithing_table" "(" <expresion> "," <tipo_dato> ")"

// Operación de Tamaño
<operacion_tamano> ::= "experience_bar" "(" <expresion> ")"
```

**Explicación simple:** Esta sección define cómo se maneja la memoria y el alcance de las variables:

- Alcance usando bloques de código con su propio contexto
- Coerción de tipos usando casting con paréntesis o la función smithing_table
- Medición de tamaño usando .^{ex}perience_bar"

### 0.3.16. Ejemplos de Uso de Manejo de Memoria y Alcance

```
// Ejemplo de alcance
{
    emerald contador = diamond(0);
```

```
    enchantment_table emerald incrementar() {
        contador = contador + 1;
        totem_undying(contador);
    }
}
// Ejemplo de coerción de tipos
emerald entero = diamond(42);
gold_nugget decimal = (gold_nugget)entero;  // Método 1
gold_nugget decimal2 = smithing_table(entero, gold_nugget);  // Método 2
// Ejemplo de operación de tamaño
bundle[5] inventario = minecart[1, 2, 3, 4, 5];
emerald tamano = experience_bar(inventario);  // Obtiene 5
// Ejemplo de anidamiento de bloques y alcance
{
    emerald x = diamond(10);
    {
        emerald y = diamond(20);
        emerald z = x + y;  // z = 30, accede a x del bloque exterior
    }
    // y no es accesible aquí
}
// x no es accesible aquí
```

## 0.4. Conjunto de pruebas

Desarrollo de las pruebas

## 0.5. Conclusiones

Aquí van las conclusiones.