



Instituto Tecnológico de Costa Rica

Escuela de Computación

# Compiladores e Intérpretes, IC5701

Etapas dos: Analizador Sintáctico (Parser)

## **Estudiantes:**

Samir Cabrera Tabash, 2022161229

Luis Urbina Salazar, 2023156802

Primer semestre del año 2025

# Índice general

Índice general	1
<b>1. Gramatica del Lenguaje</b>	<b>3</b>
1.1. Estructura Básica del Programa - Gramática BNF . . . . .	3
1.2. Sistemas de Asignación - Gramática BNF . . . . .	8
1.3. Tipos de Datos - Gramática BNF . . . . .	12
1.4. Literales - Gramática BNF . . . . .	19
1.5. Sistemas de Acceso - Gramática BNF . . . . .	28
1.6. Operadores - Gramática BNF . . . . .	34
1.7. Estructuras de Control - Gramática BNF . . . . .	40
1.8. Funciones y Procedimientos - Gramática BNF . . . . .	47
1.9. Elementos Auxiliares - Gramática BNF . . . . .	52
<b>2. Documentacion del Parser</b>	<b>58</b>
2.1. Documentacion inicial . . . . .	58
2.2. Gramática del Parser . . . . .	58
2.2.1. Resultados de GikGram . . . . .	68
2.2.2. Clase Parser . . . . .	69
2.2.3. Funciones auxiliares . . . . .	70
2.2.4. Características relevantes . . . . .	71
2.2.5. SpecialTokens . . . . .	71
2.2.6. TokenMap . . . . .	72
2.2.7. GLadosDerechos . . . . .	72
2.2.8. GNombresTerminales . . . . .	72
2.2.9. Gramatica . . . . .	72
2.2.10. Tabla Follows . . . . .	73
2.2.11. Tabla Parsing . . . . .	73
2.3. Resultados . . . . .	73

SE DEBE DE SABER QUE EN ESTE ARCHIVO  
HAY DOS GRAMATICAS, LA GRAMATICA PREVIAMENTE  
DEFINIDA EN LA ETAPA CERO Y EN LA ETAPA  
UNO, Y LA GRAMATICA DEFINIDA PARA EL PROCESAMIENTO  
DEL LENGUAJE, ASIGNACION DE ESTA ETAPA,  
ETAPA DOS. POR LO QUE PARA REVISAR LA GRAMATICA  
SE DEBE DE AVANZAR A LA PAGINA: 59

# Capítulo 1

## Gramatica del Lenguaje

### 1.1. Estructura Básica del Programa - Gramática BNF

La gramática BNF (Backus-Naur Form) para los elementos básicos que constituyen la estructura de un programa en Notch Engine se define a continuación.

#### Estructura del título del programa

El título del programa define el nombre del nuevo mundo que se crea en Notch Engine.

```
<programa> ::= <título-programa> <secciones> <punto-entrada> "worldSave"  
<título-programa> ::= "WorldName" <identificador> ":"
```

#### Secciones del programa

Un programa en Notch Engine puede contener varias secciones opcionales que deben aparecer en un orden específico.

```
<secciones> ::= <seccion-constantes> <seccion-tipos> <seccion-variables>  
               <seccion-prototipos> <seccion-rutinas>  
<secciones> ::= <seccion-constantes> <seccion-tipos> <seccion-variables>  
               <seccion-prototipos>  
<secciones> ::= <seccion-constantes> <seccion-tipos> <seccion-variables>  
               <seccion-rutinas>  
<secciones> ::= <seccion-constantes> <seccion-tipos> <seccion-prototipos>  
               <seccion-rutinas>
```

```

<secciones> ::= <seccion-constantes> <seccion-variables> <seccion-prototipos>
               <seccion-rutinas>
<secciones> ::= <seccion-tipos> <seccion-variables> <seccion-prototipos>
               <seccion-rutinas>
<secciones> ::= <seccion-constantes> <seccion-tipos> <seccion-variables>
<secciones> ::= <seccion-constantes> <seccion-tipos> <seccion-prototipos>
<secciones> ::= <seccion-constantes> <seccion-tipos> <seccion-rutinas>
<secciones> ::= <seccion-constantes> <seccion-variables> <seccion-prototipos>
<secciones> ::= <seccion-constantes> <seccion-variables> <seccion-rutinas>
<secciones> ::= <seccion-constantes> <seccion-prototipos> <seccion-rutinas>
<secciones> ::= <seccion-tipos> <seccion-variables> <seccion-prototipos>
<secciones> ::= <seccion-tipos> <seccion-variables> <seccion-rutinas>
<secciones> ::= <seccion-tipos> <seccion-prototipos> <seccion-rutinas>
<secciones> ::= <seccion-variables> <seccion-prototipos> <seccion-rutinas>
<secciones> ::= <seccion-constantes> <seccion-tipos>
<secciones> ::= <seccion-constantes> <seccion-variables>
<secciones> ::= <seccion-constantes> <seccion-prototipos>
<secciones> ::= <seccion-constantes> <seccion-rutinas>
<secciones> ::= <seccion-tipos> <seccion-variables>
<secciones> ::= <seccion-tipos> <seccion-prototipos>
<secciones> ::= <seccion-tipos> <seccion-rutinas>
<secciones> ::= <seccion-variables> <seccion-prototipos>
<secciones> ::= <seccion-variables> <seccion-rutinas>
<secciones> ::= <seccion-prototipos> <seccion-rutinas>
<secciones> ::= <seccion-constantes>
<secciones> ::= <seccion-tipos>
<secciones> ::= <seccion-variables>
<secciones> ::= <seccion-prototipos>
<secciones> ::= <seccion-rutinas>
<secciones> ::=

```

## Sección constantes (Bedrock)

La sección de constantes, identificada por la palabra clave **Bedrock**, define valores inmutables que no pueden ser alterados durante la ejecución del programa.

```

<seccion-constantes> ::= "Bedrock" <lista-constantes>
<seccion-constantes> ::= "Bedrock"

<lista-constantes> ::= <declaracion-constante> <lista-constantes>
<lista-constantes> ::= <declaracion-constante>

```

### Sección de tipos (ResourcePack)

La sección de tipos, identificada por la palabra clave **ResourcePack**, define los tipos de datos y conversiones que pueden ser utilizados en el programa.

```
<seccion-tipos> ::= "ResourcePack" <lista-tipos>
<seccion-tipos> ::= "ResourcePack"
```

```
<lista-tipos> ::= <declaracion-tipo> <lista-tipos>
<lista-tipos> ::= <declaracion-tipo>
```

### Sección de variables (Inventory)

La sección de variables, identificada por la palabra clave **Inventory**, define y opcionalmente inicializa las variables que se utilizarán en el programa.

```
<seccion-variables> ::= "Inventory" <lista-variables>
<seccion-variables> ::= "Inventory"
```

```
<lista-variables> ::= <declaracion-variable> <lista-variables>
<lista-variables> ::= <declaracion-variable>
```

### Sección de prototipos (Recipe)

La sección de prototipos, identificada por la palabra clave **Recipe**, define las declaraciones anticipadas de funciones y procedimientos que se utilizarán en el programa.

```
<seccion-prototipos> ::= "Recipe" <lista-prototipos>
<seccion-prototipos> ::= "Recipe"
```

```
<lista-prototipos> ::= <prototipo> <lista-prototipos>
<lista-prototipos> ::= <prototipo>
```

```
<prototipo> ::= <prototipo-funcion>
<prototipo> ::= <prototipo-procedimiento>
```

```
<prototipo-funcion> ::= "Spell" <identificador> "(" <lista-parametros> ")"
                                     "->" <tipo> ";"
<prototipo-procedimiento> ::= "Ritual" <identificador>
                                     "(" <lista-parametros> ")" ";"
```

## Sección de rutinas (CraftingTable)

La sección de rutinas, identificada por la palabra clave `CraftingTable`, contiene las implementaciones completas de funciones y procedimientos.

```
<seccion-rutinas> ::= "CraftingTable" <lista-rutinas>
<seccion-rutinas> ::= "CraftingTable"

<lista-rutinas> ::= <rutina> <lista-rutinas>
<lista-rutinas> ::= <rutina>

<rutina> ::= <funcion>
<rutina> ::= <procedimiento>

<funcion> ::= "Spell" <identificador> "(" <lista-parametros> ")"
           "->" <tipo> <bloque>
<procedimiento> ::= "Ritual" <identificador> "(" <lista-parametros> ")"
                  <bloque>
```

## Punto de entrada del programa (SpawnPoint)

El punto de entrada, identificado por la palabra clave `SpawnPoint`, define donde comienza la ejecución del programa.

```
<punto-entrada> ::= "SpawnPoint" <bloque>
<bloque> ::= "PolloCrudo" <lista-instrucciones> "PolloAsado"
<bloque> ::= "PolloCrudo" "PolloAsado"

<lista-instrucciones> ::= <instruccion> <lista-instrucciones>
<lista-instrucciones> ::= <instruccion>
```

## Ejemplo completo de estructura básica

A continuación se muestra un ejemplo de la estructura básica de un programa en Notch Engine usando la gramática BNF definida:

```
WorldName MiMundo:
```

```
Bedrock
```

```
Obsidian Stack MAX_NIVEL 100;
Obsidian Spider NOMBRE "Notch Engine";
```

ResourcePack

```
Anvil Stack -> Spider;  
Anvil Ghast -> Stack;
```

Inventory

```
Stack nivel = 1;  
Spider mensaje = "Bienvenido";
```

Recipe

```
Spell calcularDanio(Stack :: nivel, arma; Ghast ref modificador) -> Stack;  
Ritual mostrarEstado(Spider :: nombre; Stack nivel, salud);
```

CraftingTable

```
Spell calcularDanio(Stack :: nivel, arma; Ghast ref modificador) -> Stack  
PolloCrudo  
    Stack danioBase = nivel * 2 + 5;  
    respawn danioBase;  
PolloAsado
```

```
Ritual mostrarEstado(Spider :: nombre; Stack nivel, salud)
```

```
PolloCrudo  
    dropperSpider("Nombre: " + nombre);  
    dropperStack(nivel);  
    dropperStack(salud);  
PolloAsado
```

SpawnPoint

```
Stack miNivel = hopperStack();  
Stack miDanio = calcularDanio(miNivel, 1, 1.5);  
dropperStack(miDanio);
```

worldSave



## 1.2. Sistemas de Asignación - Gramática BNF

Esta sección define la gramática BNF para los sistemas de asignación en Notch Engine, incluyendo la asignación de constantes, tipos y variables.

### Sistema de asignación de constantes (Obsidian)

En Notch Engine, las constantes se asignan utilizando la palabra clave **Obsidian**, evocando uno de los materiales más duros de Minecraft. Una vez definidas, estas constantes no pueden ser modificadas durante la ejecución del programa.

```
<declaracion-constante> ::= "Obsidian" <tipo> <identificador>
                             <expresion-constante> ";"

<expresion-constante> ::= <literal>
<expresion-constante> ::= <identificador>
<expresion-constante> ::= <expresion-constante-aritmetica>
<expresion-constante> ::= <expresion-constante-logica>
<expresion-constante> ::= <expresion-constante-string>

<expresion-constante-aritmetica> ::= <expresion-constante> "+"
                                     <expresion-constante>
<expresion-constante-aritmetica> ::= <expresion-constante> "-"
                                     <expresion-constante>
<expresion-constante-aritmetica> ::= <expresion-constante> "*"
                                     <expresion-constante>
<expresion-constante-aritmetica> ::= <expresion-constante> "/"
                                     <expresion-constante>
<expresion-constante-aritmetica> ::= <expresion-constante> "%"
                                     <expresion-constante>
<expresion-constante-aritmetica> ::= "(" <expresion-constante-aritmetica> ")"

<expresion-constante-logica> ::= <expresion-constante> "and"
                                   <expresion-constante>
<expresion-constante-logica> ::= <expresion-constante> "or"
                                   <expresion-constante>
<expresion-constante-logica> ::= "not" <expresion-constante>
<expresion-constante-logica> ::= <expresion-constante> "xor"
                                   <expresion-constante>
<expresion-constante-logica> ::= "(" <expresion-constante-logica> ")"
```

```
<expresion-constante-string> ::= <expresion-constante> "bind"  
                                <expresion-constante>
```

Ejemplos:

```
Obsidian Stack MAX_NIVEL 100;  
Obsidian Spider MENSAJE "Bienvenido a" bind "Notch Engine";  
Obsidian Torch DEBUG_MODE On;  
Obsidian Stack CANTIDAD_INICIAL 5 * 2;
```

### Sistema de asignación de tipos (Anvil)

La asignación de tipos en Notch Engine se realiza mediante la palabra clave `Anvil`, simbolizando cómo un yunque (anvil) en Minecraft modifica y combina ítems. Esto permite definir reglas de conversión entre diferentes tipos de datos.

```
<declaracion-tipo> ::= "Anvil" <tipo-origen> "->" <tipo-destino> ";"  
<declaracion-tipo> ::= "Anvil" <tipo-origen> "->" <tipo-destino>  
                        <modo-conversion> ";"
```

```
<tipo-origen> ::= <tipo>  
<tipo-destino> ::= <tipo>
```

```
<modo-conversion> ::= "truncate"  
<modo-conversion> ::= "round"  
<modo-conversion> ::= "safe"
```

```
<tipo> ::= "Stack"  
<tipo> ::= "Rune"  
<tipo> ::= "Spider"  
<tipo> ::= "Torch"  
<tipo> ::= "Chest"  
<tipo> ::= "Book"  
<tipo> ::= "Ghast"  
<tipo> ::= "Shelf"  
<tipo> ::= "Entity"  
<tipo> ::= <identificador>
```

Ejemplos:

```
Anvil Ghast -> Stack truncate;
Anvil Stack -> Spider;
Anvil Rune -> Stack safe;
```

## Sistema de declaración de variables

La declaración de variables en Notch Engine utiliza el tipo de dato seguido por un identificador y, opcionalmente, un valor inicial. También permite la declaración múltiple de variables del mismo tipo.

```
<declaracion-variable> ::= <tipo> <identificador> "=" <expresion> ";"
<declaracion-variable> ::= <tipo> <identificador> ";"
<declaracion-variable> ::= <tipo> <lista-identificadores> ";"

<lista-identificadores> ::= <identificador> "," <lista-identificadores>
<lista-identificadores> ::= <identificador>

<tipo-complejo> ::= "Shelf" "[" <expresion-entera> "]" <tipo>
<tipo-complejo> ::= "Entity" <identificador> <lista-campos>
<tipo-complejo> ::= "Entity" <identificador>

<lista-campos> ::= <declaracion-campo> <lista-campos>
<lista-campos> ::= <declaracion-campo>

<declaracion-campo> ::= <tipo> <identificador> ";"
```

Ejemplos:

```
Stack nivel = 1;
Spider nombre = "Steve";
Torch activo;
Ghast salud = 20.5, energia = 100.0;
Shelf[10] inventario;
```

```
Entity Jugador {
    Spider nombre;
    Stack nivel;
    Ghast salud;
};
```

```
Entity Jugador steve;
```

## Ejemplos completos de sistemas de asignación

A continuación se presentan ejemplos completos que demuestran el uso de los sistemas de asignación en Notch Engine:

WorldName SistemasDeAsignacion:

### Bedrock

```
* Declaración de constantes usando Obsidian *$
Obsidian Stack MAX_NIVEL 100;
Obsidian Stack MIN_NIVEL 1;
Obsidian Spider TITULO "Notch Engine";
Obsidian Spider VERSION "1.0";
Obsidian Spider MENSAJE TITULO bind " v" bind VERSION;
Obsidian Torch DEBUG_MODE Off;
Obsidian Ghast PI 3.14159;
```

### ResourcePack

```
* Reglas de conversión de tipos usando Anvil *$
Anvil Ghast -> Stack truncate;
Anvil Stack -> Ghast;
Anvil Stack -> Spider;
Anvil Rune -> Stack safe;
Anvil Spider -> Rune;
```

### Inventory

```
* Declaración de variables simples *$
Stack nivel = 1;
Spider nombre = "Steve";
Torch activo = On;

* Declaración múltiple de variables *$
Stack x = 10, y = 20, z = 30;

* Declaración de arreglo *$
Shelf[5] items;

* Declaración de estructura *$
Entity Jugador
PolloCrudo
```

```

    Spider nombre;
    Stack nivel;
    Ghast salud;
    Torch activo;
    PolloAsado;

    Entity Jugador steve;

```

```

SpawnPoint
    $* Código principal aquí *$
    dropperSpider(MENSAJE);

```

```

worldSave

```

### 1.3. Tipos de Datos - Gramática BNF

Esta sección define la gramática BNF para los diferentes tipos de datos disponibles en Notch Engine, desde tipos atómicos básicos hasta estructuras de datos complejas.

#### Tipo de dato entero (Stack)

El tipo de dato entero se representa mediante la palabra clave **Stack**, evocando cómo los objetos se pueden apilar en Minecraft. Los enteros almacenan valores numéricos completos, sin parte decimal.

```

<tipo-entero> ::= "Stack"
<expresion-entera> ::= <literal-entero>
<expresion-entera> ::= <identificador>
<expresion-entera> ::= <expresion-aritmetica-enteros>
<expresion-entera> ::= "(" <expresion-entera> ")"

<expresion-aritmetica-enteros> ::= <expresion-entera> "+" <expresion-entera>
<expresion-aritmetica-enteros> ::= <expresion-entera> "-" <expresion-entera>
<expresion-aritmetica-enteros> ::= <expresion-entera> "*" <expresion-entera>
<expresion-aritmetica-enteros> ::= <expresion-entera> "/" <expresion-entera>
<expresion-aritmetica-enteros> ::= <expresion-entera> "%" <expresion-entera>

```

### Tipo de dato caracter (Rune)

El tipo de dato carácter se representa mediante la palabra clave **Rune**, haciendo referencia a las runas utilizadas en los encantamientos de Minecraft. Un carácter almacena un único símbolo.

```
<tipo-caracter> ::= "Rune"
<expresion-caracter> ::= <literal-caracter>
<expresion-caracter> ::= <identificador>
<expresion-caracter> ::= <operacion-caracter>
<expresion-caracter> ::= "(" <expresion-caracter> ")"

<operacion-caracter> ::= "etchUp" "(" <expresion-caracter> ")"
<operacion-caracter> ::= "etchDown" "(" <expresion-caracter> ")"
```

### Tipo de dato string (Spider)

El tipo de dato cadena de texto se representa mediante la palabra clave **Spider**, haciendo referencia a cómo las arañas en Minecraft dejan hilos (strings) cuando mueren. Las cadenas almacenan secuencias de caracteres.

```
<tipo-string> ::= "Spider"
<expresion-string> ::= <literal-string>
<expresion-string> ::= <identificador>
<expresion-string> ::= <operacion-string>
<expresion-string> ::= "(" <expresion-string> ")"

<operacion-string> ::= "bind" "(" <expresion-string> ","
                        <expresion-string> ")"
<operacion-string> ::= "#" "(" <expresion-string> ")"
<operacion-string> ::= "from" <expresion-string> "##"
                        <expresion-entera> "##" <expresion-entera>
<operacion-string> ::= "except" <expresion-string> "##" <expresion-entera>
                        "##" <expresion-entera>
<operacion-string> ::= "seek" "(" <expresion-string> ","
                        <expresion-string> ")"
```

### Tipo de dato booleano (Torch)

El tipo de dato booleano se representa mediante la palabra clave **Torch**, simbolizando cómo una antorcha en Minecraft puede estar encendida o apagada. Los booleanos representan valores de verdad: verdadero (**On**) o falso (**Off**).

```

<tipo-booleano> ::= "Torch"
<expresion-booleana> ::= <literal-booleano>
<expresion-booleana> ::= <identificador>
<expresion-booleana> ::= <operacion-logica>
<expresion-booleana> ::= <operacion-comparacion>
<expresion-booleana> ::= <operacion-caracter-bool>
<expresion-booleana> ::= "(" <expresion-booleana> ")"

<operacion-logica> ::= <expresion-booleana> "and" <expresion-booleana>
<operacion-logica> ::= <expresion-booleana> "or" <expresion-booleana>
<operacion-logica> ::= "not" <expresion-booleana>
<operacion-logica> ::= <expresion-booleana> "xor" <expresion-booleana>

<operacion-caracter-bool> ::= "isEngraved" "(" <expresion-caracter> ")"
<operacion-caracter-bool> ::= "isInscribed" "(" <expresion-caracter> ")"

```

### Tipo de dato conjunto (Chest)

El tipo de dato conjunto se representa mediante la palabra clave **Chest**, evocando cómo un cofre en Minecraft almacena múltiples objetos únicos. Los conjuntos agrupan elementos sin repetición y sin un orden específico.

```

<tipo-conjunto> ::= "Chest"
<expresion-conjunto> ::= <literal-conjunto>
<expresion-conjunto> ::= <identificador>
<expresion-conjunto> ::= <operacion-conjunto>
<expresion-conjunto> ::= "(" <expresion-conjunto> ")"

<operacion-conjunto> ::= "add" "(" <expresion-conjunto> ","
                        <expresion> ")"
<operacion-conjunto> ::= "drop" "(" <expresion-conjunto>
                        "," <expresion> ")"
<operacion-conjunto> ::= "items" "(" <expresion-conjunto> ","
                        <expresion-conjunto> ")"
<operacion-conjunto> ::= "feed" "(" <expresion-conjunto> ","
                        <expresion-conjunto> ")"
<operacion-conjunto> ::= "map" "(" <expresion-conjunto> ","
                        <expresion> ")"
<operacion-conjunto> ::= "biom" "(" <expresion-conjunto> ")"
<operacion-conjunto> ::= "kill" "(" <expresion-conjunto> ")"

```

## Tipo de dato archivo de texto (Book)

El tipo de dato archivo de texto se representa mediante la palabra clave **Book**, simbolizando un libro en Minecraft. Los archivos permiten almacenar y recuperar información persistente entre ejecuciones del programa.

```
<tipo-archivo> ::= "Book"
<expresion-archivo> ::= <literal-archivo>
<expresion-archivo> ::= <identificador>
<expresion-archivo> ::= <operacion-archivo>
<expresion-archivo> ::= "(" <expresion-archivo> ")"

<operacion-archivo> ::= "unlock" "(" <expresion-archivo> ")"
<operacion-archivo> ::= "lock" "(" <expresion-archivo> ")"
<operacion-archivo> ::= "craft" "(" <expresion-archivo> ")"
<operacion-archivo> ::= "gather" "(" <expresion-archivo> ")"
<operacion-archivo> ::= "forge" "(" <expresion-archivo> ","
                                <expresion-string> ")"
<operacion-archivo> ::= "tag" "(" <expresion-archivo> ","
                                <expresion-archivo> ")"
```

## Tipo de datos números flotantes (Ghast)

El tipo de dato para números con punto flotante se representa mediante la palabra clave **Ghast**, evocando cómo los Ghast en Minecraft flotan en el aire. Los números flotantes almacenan valores numéricos con parte decimal.

```
<tipo-flotante> ::= "Ghast"
<expresion-flotante> ::= <literal-flotante>
<expresion-flotante> ::= <identificador>
<expresion-flotante> ::= <expresion-aritmetica-flotante>
<expresion-flotante> ::= "(" <expresion-flotante> ")"

<expresion-aritmetica-flotante> ::= <expresion-flotante> ":" "+"
                                <expresion-flotante>
<expresion-aritmetica-flotante> ::= <expresion-flotante> ":" "-"
                                <expresion-flotante>
<expresion-aritmetica-flotante> ::= <expresion-flotante> ":" "*"
                                <expresion-flotante>
<expresion-aritmetica-flotante> ::= <expresion-flotante> ":" "/"
                                <expresion-flotante>
```



```
<expresion-aritmetica-flotante> ::= <expresion-flotante> ":"  
                                     <expresion-flotante>
```

### Tipo de dato arreglos (Shelf)

El tipo de dato arreglo se representa mediante la palabra clave **Shelf**, evocando una estantería en Minecraft donde cada libro (dato) ocupa un lugar específico. Los arreglos almacenan colecciones ordenadas de elementos del mismo tipo, accesibles por índice.

```
<tipo-arreglo> ::= "Shelf" "[" <expresion-entera> "]" <tipo>  
<expresion-arreglo> ::= <literal-arreglo>  
<expresion-arreglo> ::= <identificador>  
<expresion-arreglo> ::= <expresion-acceso-arreglo>  
<expresion-arreglo> ::= "(" <expresion-arreglo> ")"  
  
<expresion-acceso-arreglo> ::= <identificador> "[" <expresion-entera> "]"
```

### Tipo de dato registros (Entity)

El tipo de dato registro se representa mediante la palabra clave **Entity**, evocando cómo una entidad en Minecraft encapsula múltiples atributos y comportamientos. Los registros agrupan campos de diferentes tipos bajo un mismo nombre.

```
<tipo-registro> ::= "Entity" <identificador>  
<tipo-registro-def> ::= "Entity" <identificador> <lista-campos>  
  
<lista-campos> ::= <campo> <lista-campos>  
<lista-campos> ::= <campo>  
  
<campo> ::= <tipo> <identificador> ";"  
  
<expresion-registro> ::= <literal-registro>  
<expresion-registro> ::= <identificador>  
<expresion-registro> ::= <expresion-acceso-registro>  
<expresion-registro> ::= "(" <expresion-registro> ")"  
  
<expresion-acceso-registro> ::= <identificador> "@" <identificador>
```

## Ejemplos completos de tipos de datos

A continuación se presentan ejemplos que demuestran el uso de los diferentes tipos de datos en Notch Engine:

WorldName TiposDatos:

Inventory

```
$* Tipos básicos *$
Stack contador = 0;
Rune inicial = 'A';
Spider nombre = "Steve";
Torch activo = 0n;
Chest vocales = { 'a', 'e', 'i', 'o', 'u' :};
Book registro = {/ "log.txt", 'E' /};
Ghast temperatura = 36.5;
```

```
$* Tipos compuestos *$
Shelf[10] inventario;
```

Entity Jugador

PolloCrudo

```
    Spider nombre;
    Stack nivel;
    Ghast salud;
    Torch activo;
```

PolloAsado;

Entity Jugador steve;

SpawnPoint

PolloCrudo

```
$* Operaciones con enteros *$
Stack a = 5 + 3;
Stack b = a * 2;
Stack c = b // 3;
```

```
$* Operaciones con caracteres *$
Rune letra = 'a';
Rune mayuscula = etchUp(letra);
Torch esLetra = isEngraved(letra);
```

```

$* Operaciones con strings *$
Spider saludo = "Hola";
Spider mensaje = bind(saludo, " Mundo");
Stack longitud = #(mensaje);
Spider subcadena = from mensaje ## 0 ## 4;

$* Operaciones con booleanos *$
Torch condicion1 = On;
Torch condicion2 = Off;
Torch resultado = condicion1 and condicion2;

$* Operaciones con conjuntos *$
Chest conjunto1 = {: 1, 2, 3 :};
add(conjunto1, 4);
Chest conjunto2 = {: 3, 4, 5 :};
Chest union = items(conjunto1, conjunto2);

$* Operaciones con archivos *$
unlock(registro);
forge(registro, "Entrada de log");
lock(registro);

$* Operaciones con flotantes *$
Ghast pi = 3.14159;
Ghast doble = pi :* 2.0;

$* Operaciones con arreglos *$
inventario[0] = 5;
Stack valor = inventario[0];

$* Operaciones con registros *$
steve@nombre = "Steve";
steve@nivel = 1;
steve@salud = 20.0;
steve@activo = On;

Spider nombreJugador = steve@nombre;
PolloAsado

```

worldSave

## 1.4. Literales - Gramática BNF

Esta sección define la gramática BNF para los diferentes tipos de valores literales en Notch Engine. Los literales son representaciones directas de datos en el código fuente del programa.

### Literales booleanas (On/Off)

Las literales booleanas representan valores de verdad: verdadero (**On**) o falso (**Off**), evocando el estado de una palanca o interruptor en Minecraft.

```
<literal-booleano> ::= "On"  
<literal-booleano> ::= "Off"
```

Ejemplos:

```
Torch activo = On;  
Torch inactivo = Off;
```

### Literales de conjuntos ({: ... :})

Las literales de conjuntos representan colecciones de elementos únicos sin un orden específico. Se delimitan con los símbolos {: y :}.

```
<literal-conjunto> ::= "{:" <lista-elementos> ":}"  
<literal-conjunto> ::= "{:" ":}"  
  
<lista-elementos> ::= <expresion> "," <lista-elementos>  
<lista-elementos> ::= <expresion>
```

Ejemplos:

```
Chest numeros = {: 1, 2, 3, 4, 5 :};  
Chest letras = {: 'a', 'e', 'i', 'o', 'u' :};  
Chest vacio = {: :};
```

### Literales de archivos ({/ ... /})

Las literales de archivos especifican un nombre de archivo y un modo de acceso. El modo puede ser 'L' para lectura, 'E' para escritura, o 'A' para actualización.

```
<literal-archivo> ::= "{/" <literal-string> "," <literal-caracter> "/"
```

```
<modo-archivo> ::= "'L'"
```

```
<modo-archivo> ::= "'E'"
```

```
<modo-archivo> ::= "'A'"
```

Ejemplos:

```
Book archivo = {/ "datos.txt", 'L' /};  
Book registro = {/ "log.txt", 'E' /};  
Book configuracion = {/ "config.ini", 'A' /};
```

### Literales de números flotantes (-3.14)

Las literales de números flotantes representan valores numéricos con parte decimal.

```
<literal-flotante> ::= <numero-entero> "." <numero-entero>
```

```
<literal-flotante> ::= <numero-entero> "."
```

```
<literal-flotante> ::= "." <numero-entero>
```

```
<literal-flotante> ::= "-" <literal-flotante>
```

Ejemplos:

```
Ghast pi = 3.14159;  
Ghast temperatura = 36.6;  
Ghast negativo = -2.5;  
Ghast decimal = .5;
```

### Literales de enteros (5, -5)

Las literales de enteros representan valores numéricos completos, sin parte decimal.

```
<literal-entero> ::= <numero-entero>
```

```
<literal-entero> ::= "-" <numero-entero>
```

```

<numero-entero> ::= <digito> <resto-numero>
<numero-entero> ::= <digito>

<resto-numero> ::= <digito> <resto-numero>
<resto-numero> ::= <digito>

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

```

Ejemplos:

```

Stack positivo = 42;
Stack negativo = -7;
Stack cero = 0;

```

## Literales de caracteres ('K')

Las literales de caracteres representan un único símbolo, encerrado entre comillas simples.

```

<literal-caracter> ::= "'" <caracter> "'"

<caracter> ::= <letra>
<caracter> ::= <digito>
<caracter> ::= <simbolo>
<caracter> ::= <escape-secuencia>

<escape-secuencia> ::= "\n"
<escape-secuencia> ::= "\t"
<escape-secuencia> ::= "\r"
<escape-secuencia> ::= "\\"
<escape-secuencia> ::= "\'"
<escape-secuencia> ::= "\""

```

```
<letra> ::= "a"  
<letra> ::= "b"  
<letra> ::= "c"  
<letra> ::= "d"  
<letra> ::= "e"  
<letra> ::= "f"  
<letra> ::= "g"  
<letra> ::= "h"  
<letra> ::= "i"  
<letra> ::= "j"  
<letra> ::= "k"  
<letra> ::= "l"  
<letra> ::= "m"  
<letra> ::= "n"  
<letra> ::= "o"  
<letra> ::= "p"  
<letra> ::= "q"  
<letra> ::= "r"  
<letra> ::= "s"  
<letra> ::= "t"  
<letra> ::= "u"  
<letra> ::= "v"  
<letra> ::= "w"  
<letra> ::= "x"  
<letra> ::= "y"  
<letra> ::= "z"  
<letra> ::= "A"  
<letra> ::= "B"  
<letra> ::= "C"  
<letra> ::= "D"  
<letra> ::= "E"  
<letra> ::= "F"  
<letra> ::= "G"  
<letra> ::= "H"  
<letra> ::= "I"  
<letra> ::= "J"  
<letra> ::= "K"  
<letra> ::= "L"  
<letra> ::= "M"  
<letra> ::= "N"  
<letra> ::= "O"
```

```

<letra> ::= "P"
<letra> ::= "Q"
<letra> ::= "R"
<letra> ::= "S"
<letra> ::= "T"
<letra> ::= "U"
<letra> ::= "V"
<letra> ::= "W"
<letra> ::= "X"
<letra> ::= "Y"
<letra> ::= "Z"

```

```

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

```



```

<simbolo> ::= ">"
<simbolo> ::= "~"
<simbolo> ::= "'"

```

Ejemplos:

```

Rune letra = 'A';
Rune digito = '5';
Rune simbolo = '@';
Rune nuevaLinea = '\n';

```

## Literales de strings ("string")

Las literales de strings representan secuencias de caracteres, encerradas entre comillas dobles.

```

<literal-string> ::= "\"" <secuencia-caracteres> "\""
<literal-string> ::= "\"\""
<secuencia-caracteres> ::= <caracter> <secuencia-caracteres>
<secuencia-caracteres> ::= <caracter>

```

Ejemplos:

```

Spider nombre = "Steve";
Spider mensaje = "Bienvenido a Notch Engine";
Spider vacio = "";
Spider conEscape = "Línea 1\nLínea 2";

```

## Literales de arreglos ([1, 2, 3, 4, 5])

Las literales de arreglos representan colecciones ordenadas de elementos del mismo tipo, encerradas entre corchetes.

```

<literal-arreglo> ::= "[" <lista-elementos> "]"
<literal-arreglo> ::= "[" "]"
<lista-elementos> ::= <expresion> "," <lista-elementos>
<lista-elementos> ::= <expresion>

```

Ejemplos:

```

Shelf[5] numeros = [1, 2, 3, 4, 5];
Shelf[3] nombres = ["Steve", "Alex", "Herobrine"];
Shelf[0] vacio = [];

```

### Literales de registros ({<id>: <value>, <id>: <value>, ...})

Las literales de registros representan estructuras que agrupan campos de diferentes tipos, con cada campo identificado por un nombre.

```
<literal-registro> ::= <lista-campos-valor>
```

```
<literal-registro> ::= "{" "}"
```

```
<lista-campos-valor> ::= <campo-valor> "," <lista-campos-valor>
```

```
<lista-campos-valor> ::= <campo-valor>
```

```
<campo-valor> ::= <identificador> ":" <expresion>
```

Ejemplos:

```
Entity Jugador steve = {  
    nombre: "Steve",  
    nivel: 1,  
    salud: 20.0,  
    activo: On  
};
```

```
Entity Posicion punto = {x: 10, y: 20, z: 30};
```

```
Entity Config configuracion = {};
```

### Identificadores válidos

Un identificador es un nombre que se utiliza para referirse a variables, constantes, tipos, funciones, etc.

```
<identificador> ::= <letra> <resto-identificador>
```

```
<identificador> ::= <letra>
```

```
<resto-identificador> ::= <letra> <resto-identificador>
```

```
<resto-identificador> ::= <digito> <resto-identificador>
```

```
<resto-identificador> ::= "_" <resto-identificador>
```

```
<resto-identificador> ::= <letra>
```

```
<resto-identificador> ::= <digito>
```

```
<resto-identificador> ::= "_"
```

Ejemplos de identificadores válidos:

```
contador
nombreJugador
x
posicion_3d
_temporal
CONSTANTE
```

## Ejemplos completos de literales

A continuación se presentan ejemplos que demuestran el uso de los diferentes tipos de literales en Notch Engine:

WorldName Literales:

Inventory

```
$* Literales booleanas *$
Torch verdadero = On;
Torch falso = Off;

$* Literales de conjuntos *$
Chest numeros = {: 1, 2, 3, 4, 5 :};
Chest vocales = {: 'a', 'e', 'i', 'o', 'u' :};
Chest vacio = {: :};

$* Literales de archivos *$
Book archivoLectura = {/ "datos.txt", 'L' /};
Book archivoEscritura = {/ "salida.txt", 'E' /};
Book archivoActualizar = {/ "config.ini", 'A' /};

$* Literales de números flotantes *$
Ghast pi = 3.14159;
Ghast e = 2.71828;
Ghast negativo = -1.5;
Ghast decimal = .5;

$* Literales de enteros *$
Stack positivo = 42;
Stack negativo = -7;
Stack cero = 0;

$* Literales de caracteres *$
```

```

Rune letraMayuscula = 'A';
Rune letraMinuscula = 'z';
Rune digito = '7';
Rune simbolo = '#';
Rune nuevaLinea = '\n';

$* Literales de strings *$
Spider nombre = "Steve";
Spider mensaje = "Bienvenido a Notch Engine";
Spider lineas = "Línea 1\nLínea 2";
Spider vacio = "";

$* Literales de arreglos *$
Shelf[5] enteros = [1, 2, 3, 4, 5];
Shelf[3] cadenas = ["uno", "dos", "tres"];
Shelf[0] vacio = [];

$* Literales de registros *$
Entity Jugador PolloCrudo
    Spider nombre;
    Stack nivel;
    Ghast salud;
    Torch activo;
PolloAsado;

Entity Jugador steve
PolloCrudo
    nombre: "Steve",
    nivel: 1,
    salud: 20.0,
    activo: On
PolloAsado;

Entity Posicion punto = {x: 10, y: 20, z: 30};

```

```

SpawnPoint
    PolloCrudo
    dropperSpider("Ejemplos de literales en Notch Engine");
    PolloAsado

```

worldSave

## 1.5. Sistemas de Acceso - Gramática BNF

Esta sección define la gramática BNF para los diferentes sistemas de acceso a datos en Notch Engine, incluyendo acceso a arreglos, strings, registros y los operadores de asignación.

### Sistema de acceso arreglos ([2][3][4])

El sistema de acceso a arreglos permite obtener o modificar elementos individuales dentro de un arreglo especificando su posición mediante índices entre corchetes. Notch Engine utiliza la notación de índices múltiples consecutivos, similar a lenguajes como C y C++.

```
<acceso-arreglo> ::= <identificador> "[" <expresion-entera> "]"  
<acceso-arreglo> ::= <acceso-arreglo> "[" <expresion-entera> "]"
```

```
<expresion-acceso-arreglo> ::= <acceso-arreglo>
```

Ejemplos:

```
$* Acceso a un arreglo unidimensional *$  
Shelf[10] numeros;  
numeros[0] = 5;  
Stack valor = numeros[1];
```

```
$* Acceso a un arreglo bidimensional *$  
Shelf[3] Shelf[3] matriz;  
matriz[0][0] = 1;  
Stack elemento = matriz[1][2];
```

```
$* Acceso a un arreglo tridimensional *$  
Shelf[2] Shelf[2] Shelf[2] cubo;  
cubo[0][1][1] = 7;  
Stack valor3D = cubo[1][0][1];
```

### Sistema de acceso strings (string[1])

El sistema de acceso a strings permite obtener caracteres individuales dentro de una cadena especificando su posición mediante un índice entre corchetes, similar a como se accede a los elementos de un arreglo.

`<acceso-string> ::= <identificador> "[" <expresion-entera> "]"`

`<expresion-acceso-string> ::= <acceso-string>`

Ejemplos:

```
Spider nombre = "Steve";
```

```
Rune primeraLetra = nombre[0];  $* Obtiene 'S' *$
```

```
Rune tercerLetra = nombre[2];   $* Obtiene 'e' *$
```

\$\* También se puede usar para modificar caracteres \*\$

```
Spider palabra = "casa";
```

```
palabra[0] = 'm';                $* Ahora palabra es "masa" *$
```

### Sistema de acceso registros (@ - e.g.: registro@campo)

El sistema de acceso a registros permite obtener o modificar campos individuales dentro de un registro utilizando el operador arroba (@) entre el nombre del registro y el nombre del campo.

`<acceso-registro> ::= <identificador> "@" <identificador>`

`<expresion-acceso-registro> ::= <acceso-registro>`

Ejemplos:

```
Entity Jugador
```

```
PolloCrudo
```

```
    Spider nombre;
```

```
    Stack nivel;
```

```
    Ghast salud;
```

```
    Torch activo;
```

```
PolloAsado;
```

```
Entity Jugador steve;
```

\$\* Asignación a campos del registro \*\$

```
steve@nombre = "Steve";
```

```
steve@nivel = 1;
```

```
steve@salud = 20.0;
```

```
steve@activo = 0n;
```

```
$* Acceso a campos del registro *$
Spider nombreJugador = steve@nombre;
Stack nivelJugador = steve@nivel;
```

### Asignación y Familia (= - i.e: =, +=, -=, \*=, /=, %=)

Notch Engine proporciona una familia de operadores de asignación que permiten no solo asignar valores a variables sino también realizar operaciones combinadas de asignación y aritmética. Estos operadores son similares a los que se encuentran en lenguajes como C y C++.

```
<asignacion> ::= <lvalue> "=" <expresion>
<asignacion> ::= <lvalue> "+=" <expresion>
<asignacion> ::= <lvalue> "-=" <expresion>
<asignacion> ::= <lvalue> "*=" <expresion>
<asignacion> ::= <lvalue> "/=" <expresion>
<asignacion> ::= <lvalue> "%=" <expresion>
```

```
<lvalue> ::= <identificador>
<lvalue> ::= <acceso-arreglo>
<lvalue> ::= <acceso-string>
<lvalue> ::= <acceso-registro>
```

Ejemplos:

```
$* Asignación simple *$
Stack contador = 0;
```

```
$* Asignaciones combinadas para enteros *$
contador += 5;      $* Equivalente a: contador = contador + 5 *$
contador -= 2;      $* Equivalente a: contador = contador - 2 *$
contador *= 3;      $* Equivalente a: contador = contador * 3 *$
contador /= 2;      $* Equivalente a: contador = contador // 2 *$
contador %= 4;      $* Equivalente a: contador = contador % 4 *$
```

```
$* Asignaciones con acceso a arreglos *$
Shelf[5] numeros;
numeros[0] = 10;
numeros[1] += 5;
```

```
$* Asignaciones con acceso a strings *$
Spider texto = "abcde";
```

```
texto[0] = 'A';      $* Modifica el primer carácter *$
```

```
$* Asignaciones con acceso a registros *$  
Entity Jugador steve;  
steve@nivel = 1;  
steve@nivel += 1;    $* Incrementa el nivel *$
```

## Expresiones combinadas de acceso

Notch Engine permite combinar diferentes formas de acceso para trabajar con estructuras de datos complejas, como arreglos de registros o registros que contienen arreglos.

```
<expresion-combinada> ::= <acceso-registro> "[" <expresion-entera> "]"  
<expresion-combinada> ::= <acceso-arreglo> "@" <identificador>
```

Ejemplos:

```
$* Arreglo de registros *$  
Entity Jugador  
PolloCrudo  
    Spider nombre;  
    Stack nivel;  
PolloAsado;
```

```
Shelf[3] Entity Jugador equipo;
```

```
$* Acceso a un campo de un registro dentro de un arreglo *$  
equipo[0]@nombre = "Steve";  
equipo[1]@nombre = "Alex";  
Spider segundoJugador = equipo[1]@nombre;
```

```
$* Registro que contiene un arreglo *$  
Entity Inventario  
PolloCrudo  
    Shelf[10] Stack items;  
    Stack capacidad;  
PolloAsado;
```

```
Entity Inventario mochila;
```

```
$* Acceso a un elemento de un arreglo dentro de un registro *$
```



```

mochila@items[0] = 5;
mochila@items[1] = 10;
Stack primerItem = mochila@items[0];

```

## Ejemplos completos de sistemas de acceso

A continuación se presentan ejemplos que demuestran el uso de los diferentes sistemas de acceso en un programa Notch Engine:

WorldName SistemasAcceso:

Inventory

```

$* Declaración de tipos y variables para los ejemplos *$
Shelf[5] numeros;
Spider texto = "Notch Engine";

```

Entity Punto

```

PolloCrudo
    Stack x;
    Stack y;
    Stack z;
PolloAsado;

```

Entity Punto origen;

Entity Jugador

```

PolloCrudo
    Spider nombre;
    Stack nivel;
    Shelf[3] Stack inventario;
PolloAsado;

```

Shelf[2] Entity Jugador jugadores;

SpawnPoint

PolloCrudo

```

$* Acceso a arreglos *$
numeros[0] = 10;
numeros[1] = 20;
numeros[2] = 30;

```

```

Stack suma = numeros[0] + numeros[1];

*$ Acceso a strings *$
Rune primeraLetra = texto[0];
texto[5] = 'E';  $* Modifica el 6to carácter *$

*$ Acceso a registros *$
origen@x = 0;
origen@y = 0;
origen@z = 0;

*$ Operadores de asignación compuesta *$
numeros[0] += 5;  $* Ahora es 15 *$
numeros[1] -= 5;  $* Ahora es 15 *$
numeros[2] *= 2;  $* Ahora es 60 *$

origen@x += 10;  $* Ahora es 10 *$

*$ Acceso combinado (arreglo de registros) *$
jugadores[0]@nombre = "Steve";
jugadores[0]@nivel = 1;
jugadores[0]@inventario[0] = 5;
jugadores[0]@inventario[1] = 10;

jugadores[1]@nombre = "Alex";
jugadores[1]@nivel = 2;

*$ Uso de los valores accedidos *$
dropperSpider("Jugador 1: " bind jugadores[0]@nombre);
dropperStack(jugadores[0]@nivel);

dropperSpider("Jugador 2: " bind jugadores[1]@nombre);
dropperStack(jugadores[1]@nivel);
PolloAsado

worldSave

```

## 1.6. Operadores - Gramática BNF

Esta sección define la gramática BNF para los diferentes operadores disponibles en Notch Engine, incluyendo operadores aritméticos, de incremento, lógicos, de cadenas, conjuntos, archivos, números flotantes y comparación.

### Operaciones aritméticas básicas de enteros (+, -, \*, //, %)

Las operaciones aritméticas básicas para enteros en Notch Engine incluyen suma, resta, multiplicación, división entera y módulo (resto). La división entera se representa con dos barras diagonales (//) para distinguirla de la división flotante.

```
<expresion-aritmetica-enteros> ::= <expresion-entera> "+" <expresion-entera>
<expresion-aritmetica-enteros> ::= <expresion-entera> "-" <expresion-entera>
<expresion-aritmetica-enteros> ::= <expresion-entera> "*" <expresion-entera>
<expresion-aritmetica-enteros> ::= <expresion-entera> "/" <expresion-entera>
<expresion-aritmetica-enteros> ::= <expresion-entera> "%" <expresion-entera>
<expresion-aritmetica-enteros> ::= "(" <expresion-aritmetica-enteros> ")"
<expresion-aritmetica-enteros> ::= "-" <expresion-entera>

<expresion-entera> ::= <literal-entero>
<expresion-entera> ::= <identificador>
<expresion-entera> ::= <expresion-aritmetica-enteros>
<expresion-entera> ::= <acceso-arreglo>
<expresion-entera> ::= <acceso-registro>
<expresion-entera> ::= <llamada-funcion>
```

### Incremento y Decremento (soulsand, magma)

Las operaciones de incremento y decremento en Notch Engine se representan mediante las palabras clave **soulsand** (incremento) y **magma** (decremento), haciendo referencia a cómo estos bloques en Minecraft hacen subir o bajar al jugador cuando se combinan con agua.

```
<expresion-incremento> ::= "soulsand" <identificador>
<expresion-incremento> ::= "magma" <identificador>
<expresion-incremento> ::= "soulsand" <acceso-arreglo>
<expresion-incremento> ::= "magma" <acceso-arreglo>
<expresion-incremento> ::= "soulsand" <acceso-registro>
<expresion-incremento> ::= "magma" <acceso-registro>
```

## Operaciones básicas sobre caracteres (isEngraved, isInscribed, etchUp, etchDown)

Notch Engine proporciona operaciones específicas para manipular caracteres, incluyendo verificación de tipo y transformación de caso.

```
<operacion-caracter> ::= "isEngraved" "(" <expresion-caracter> ")"  
<operacion-caracter> ::= "isInscribed" "(" <expresion-caracter> ")"  
<operacion-caracter> ::= "etchUp" "(" <expresion-caracter> ")"  
<operacion-caracter> ::= "etchDown" "(" <expresion-caracter> ")"
```

## Operaciones lógicas solicitadas (and, or, not, xor)

Las operaciones lógicas en Notch Engine utilizan palabras en lugar de símbolos, lo que aumenta la legibilidad del código.

```
<expresion-logica> ::= <expresion-booleana> "and" <expresion-booleana>  
<expresion-logica> ::= <expresion-booleana> "or" <expresion-booleana>  
<expresion-logica> ::= "not" <expresion-booleana>  
<expresion-logica> ::= <expresion-booleana> "xor" <expresion-booleana>  
<expresion-logica> ::= "(" <expresion-logica> ")"  
  
<expresion-booleana> ::= <literal-booleano>  
<expresion-booleana> ::= <identificador>  
<expresion-booleana> ::= <expresion-logica>  
<expresion-booleana> ::= <expresion-comparacion>  
<expresion-booleana> ::= <operacion-caracter-bool>  
<expresion-booleana> ::= <acceso-arreglo>  
<expresion-booleana> ::= <acceso-registro>  
<expresion-booleana> ::= <llamada-funcion>
```

## Operaciones de Strings solicitadas (bind, #, from ##, except ##, seek)

Notch Engine proporciona operaciones específicas para manipular cadenas de texto, incluyendo concatenación, medición de longitud, extracción de subcadenas y búsqueda.

```
<operacion-string> ::= "bind" "(" <expresion-string> ","  
                        <expresion-string> ")"  
<operacion-string> ::= "#" "(" <expresion-string> ")"  
<operacion-string> ::= "from" <expresion-string> "##" <expresion-entera>
```

```

        "##" <expresion-entera>
<operacion-string> ::= "except" <expresion-string> "##" <expresion-entera>
        "##" <expresion-entera>
<operacion-string> ::= "seek" "(" <expresion-string> ","
        <expresion-string> ")"

```

### Operaciones de conjuntos solicitadas (add, drop, items, feed, map, biom, kill)

Notch Engine proporciona operaciones específicas para manipular conjuntos, incluyendo adición, eliminación, unión, intersección, pertenencia y vaciado.

```

<operacion-conjunto> ::= "add" "(" <expresion-conjunto> ","
        <expresion> ")"
<operacion-conjunto> ::= "drop" "(" <expresion-conjunto> ","
        <expresion> ")"
<operacion-conjunto> ::= "items" "(" <expresion-conjunto> ","
        <expresion-conjunto> ")"
<operacion-conjunto> ::= "feed" "(" <expresion-conjunto> ","
        <expresion-conjunto> ")"
<operacion-conjunto> ::= "map" "(" <expresion-conjunto> ","
        <expresion> ")"
<operacion-conjunto> ::= "biom" "(" <expresion-conjunto> ")"
<operacion-conjunto> ::= "kill" "(" <expresion-conjunto> ")"

```

### Operaciones de archivos solicitadas (unlock, lock, craft, gather, forge, tag)

Notch Engine proporciona operaciones específicas para manipular archivos, incluyendo apertura, cierre, creación, lectura, escritura y concatenación.

```

<operacion-archivo> ::= "unlock" "(" <expresion-archivo> ")"
<operacion-archivo> ::= "lock" "(" <expresion-archivo> ")"
<operacion-archivo> ::= "craft" "(" <expresion-archivo> ")"
<operacion-archivo> ::= "gather" "(" <expresion-archivo> ")"
<operacion-archivo> ::= "forge" "(" <expresion-archivo> ","
        <expresion-string> ")"
<operacion-archivo> ::= "tag" "(" <expresion-archivo> ","
        <expresion-archivo> ")"

```

## Operaciones de números flotantes (:+, :-, :\*, :%, ://)

Las operaciones aritméticas para números flotantes en Notch Engine utilizan los mismos símbolos que las operaciones para enteros, pero precedidos por dos puntos (:) para diferenciarlas.

```
<expresion-aritmetica-flotante> ::= <expresion-flotante> ":+"  
                                <expresion-flotante>  
<expresion-aritmetica-flotante> ::= <expresion-flotante> ":-"  
                                <expresion-flotante>  
<expresion-aritmetica-flotante> ::= <expresion-flotante> ":*"  
                                <expresion-flotante>  
<expresion-aritmetica-flotante> ::= <expresion-flotante> "://"  
                                <expresion-flotante>  
<expresion-aritmetica-flotante> ::= <expresion-flotante> ":%"  
                                <expresion-flotante>  
<expresion-aritmetica-flotante> ::= "(" <expresion-aritmetica-flotante> ")"  
<expresion-aritmetica-flotante> ::= "-" <expresion-flotante>  
  
<expresion-flotante> ::= <literal-flotante>  
<expresion-flotante> ::= <identificador>  
<expresion-flotante> ::= <expresion-aritmetica-flotante>  
<expresion-flotante> ::= <acceso-arreglo>  
<expresion-flotante> ::= <acceso-registro>  
<expresion-flotante> ::= <llamada-funcion>
```

## Operaciones de comparación solicitadas (<, >, <=, >=, is, isNot)

Las operaciones de comparación en Notch Engine permiten evaluar relaciones entre valores y producir resultados booleanos.

```
<expresion-comparacion> ::= <expresion> "<" <expresion>  
<expresion-comparacion> ::= <expresion> ">" <expresion>  
<expresion-comparacion> ::= <expresion> "<=" <expresion>  
<expresion-comparacion> ::= <expresion> ">=" <expresion>  
<expresion-comparacion> ::= <expresion> "is" <expresion>  
<expresion-comparacion> ::= <expresion> "isNot" <expresion>  
<expresion-comparacion> ::= "(" <expresion-comparacion> ")"
```

## Ejemplos completos de operadores

A continuación se presentan ejemplos que demuestran el uso de los diferentes operadores en un programa Notch Engine:

WorldName Operadores:

Inventory

```
$* Variables para los ejemplos *$
Stack a = 10;
Stack b = 5;
Stack c;
Ghast x = 3.5;
Ghast y = 1.5;
Ghast z;
Spider cadena1 = "Notch";
Spider cadena2 = "Engine";
Rune letra = 'a';
Torch condicion1 = On;
Torch condicion2 = Off;
Chest conjunto1 = {: 1, 2, 3 :};
Chest conjunto2 = {: 3, 4, 5 :};
Book archivo = {/ "datos.txt", 'E' /};
```

SpawnPoint

PolloCrudo

```
$* Operaciones aritméticas básicas de enteros *$
c = a + b;          $* c = 15 *$
c = a - b;          $* c = 5 *$
c = a * b;          $* c = 50 *$
c = a // b;         $* c = 2 *$
c = a % b;          $* c = 0 *$
```

```
$* Incremento y decremento *$
soulsand a;         $* a = 11 *$
magma b;            $* b = 4 *$
```

```
$* Operaciones básicas sobre caracteres *$
Torch esLetra = isEngraved(letra); $* On *$
Torch esDigito = isInscribed(letra); $* Off *$
Rune mayuscula = etchUp(letra);     $* 'A' *$
Rune minuscula = etchDown('B');    $* 'b' *$
```

```
$* Operaciones lógicas *$
Torch resultado1 = condicion1 and condicion2; $* Off *$
```

```

Torch resultado2 = condicion1 or condicion2;    $* On *$
Torch resultado3 = not condicion1;              $* Off *$
Torch resultado4 = condicion1 xor condicion2;    $* On *$

$* Operaciones de strings *$
Spider completo = bind(cadena1, " " bind cadena2); $* "Notch Engine" *$
Stack longitud = #(completo);                    $* 12 *$
Spider subcadena = from completo ## 0 ## 5;       $* "Notch" *$
Spider sinNombre = except completo ## 0 ## 6;     $* "Engine" *$
Stack posicion = seek(completo, "Engine");        $* 6 *$

$* Operaciones de conjuntos *$
add(conjunto1, 4);                               $* conjunto1 =
                                                    {: 1, 2, 3, 4 :} *$
drop(conjunto2, 3);                             $* conjunto2 = {: 4, 5 :} *$
Chest union = items(conjunto1, conjunto2);        $* {: 1, 2, 3, 4, 5 :} *$
Chest interseccion = feed(conjunto1, conjunto2);  $* {: 4 :} *$
Torch pertenece = map(conjunto1, 2);              $* On *$
Torch estaVacio = biom(conjunto1);               $* Off *$
kill(conjunto1);                                $* conjunto1 = {: :} *$

$* Operaciones de archivos *$
unlock(archivo);
forge(archivo, "Datos de prueba");
lock(archivo);

$* Operaciones de números flotantes *$
z = x :+ y;          $* z = 5.0 *$
z = x :- y;          $* z = 2.0 *$
z = x :* y;          $* z = 5.25 *$
z = x :// y;         $* z = 2.33... *$
z = x :% y;          $* z = 0.5 *$

$* Operaciones de comparación *$
Torch comp1 = a < b;      $* Off *$
Torch comp2 = a > b;      $* On *$
Torch comp3 = a <= b;     $* Off *$
Torch comp4 = a >= b;     $* On *$
Torch comp5 = a is b;     $* Off *$
Torch comp6 = a isNot b;  $* On *$
PolloAsado

```



worldSave

## 1.7. Estructuras de Control - Gramática BNF

Esta sección define la gramática BNF para las estructuras de control disponibles en Notch Engine, incluyendo bloques, ciclos, condicionales y control de flujo.

### Manejo de Bloques de más de una instrucción (PolloCrudo PolloAsado)

En Notch Engine, los bloques de instrucciones se delimitan con las palabras clave `PolloCrudo` y `PolloAsado`. Estos bloques permiten agrupar múltiples instrucciones para que sean tratadas como una sola unidad.

```
<bloque> ::= "PolloCrudo" <lista-instrucciones> "PolloAsado"
<bloque> ::= "PolloCrudo" "PolloAsado"

<lista-instrucciones> ::= <instruccion> <lista-instrucciones>
<lista-instrucciones> ::= <instruccion>

<instruccion> ::= <instruccion-simple> ";"
<instruccion> ::= <instruccion-control>

<instruccion-simple> ::= <asignacion>
<instruccion-simple> ::= <expresion-incremento>
<instruccion-simple> ::= <llamada-procedimiento>
<instruccion-simple> ::= <operacion-conjunto>
<instruccion-simple> ::= <operacion-archivo>
<instruccion-simple> ::= <instruccion-entrada>
<instruccion-simple> ::= <instruccion-salida>
```

### Instrucción while (repeater <cond> craft <instrucción>)

La instrucción `while` en Notch Engine se representa mediante la palabra clave `repeater`, que evoca la idea de un repetidor de redstone en Minecraft enviando señales constantemente mientras recibe energía. Esta estructura repite un bloque de código mientras una condición sea verdadera.

```
<instruccion-while> ::= "repeater" <expresion-booleana> "craft"
```

```

        <bloque>
<instruccion-while> ::= "repeater" <expresion-booleana> "craft"
        <instruccion-simple> ";"

```

### Instrucción if-then-else (target <cond> craft hit <inst> miss <inst>)

La instrucción **if-then-else** en Notch Engine se representa mediante la palabra clave **target**, evocando la idea del bloque objetivo en Minecraft que evalúa si se dio en el centro o no. Esta estructura ejecuta un bloque de código si la condición es verdadera y opcionalmente otro bloque si es falsa.

```

<instruccion-if> ::= "target" <expresion-booleana> "craft"
        "hit" <bloque>
<instruccion-if> ::= "target" <expresion-booleana> "craft"
        "hit" <bloque> "miss" <bloque>
<instruccion-if> ::= "target" <expresion-booleana> "craft"
        "miss" <bloque>
<instruccion-if> ::= "target" <expresion-booleana> "craft"
        "hit" <instruccion-simple> ";"
<instruccion-if> ::= "target" <expresion-booleana> "craft"
        "hit" <instruccion-simple> ";" "miss" <bloque>
<instruccion-if> ::= "target" <expresion-booleana> "craft"
        "hit" <bloque> "miss" <instruccion-simple> ";"
<instruccion-if> ::= "target" <expresion-booleana> "craft"
        "hit" <instruccion-simple> ";" "miss"
        <instruccion-simple> ";"
<instruccion-if> ::= "target" <expresion-booleana> "craft"
        "miss" <instruccion-simple> ";"

```

### Instrucción switch (jukebox <condition> craft, disc <case> :, silence)

La instrucción **switch** en Notch Engine se representa mediante la palabra clave **jukebox**, evocando la idea de una caja de discos en Minecraft que puede reproducir diferentes canciones. Esta estructura permite seleccionar entre múltiples bloques de código según el valor de una expresión.

```

<instruccion-switch> ::= "jukebox" <expresion> "craft" <lista-casos>
<instruccion-switch> ::= "jukebox" <expresion> "craft"

<lista-casos> ::= <caso> <lista-casos>
<lista-casos> ::= <caso>

```

```

<caso> ::= "disc" <expresion> ":" <bloque>
<caso> ::= "silence" ":" <bloque>

```

### Instrucción Repeat-until (spawnner <instrucciones> exhausted <cond>)

La instrucción **do-while** en Notch Engine se representa mediante las palabras clave **spawnner** y **exhausted**, evocando la idea de un generador de monstruos en Minecraft que continúa generando criaturas hasta que se cumple una condición. Esta estructura ejecuta un bloque de código repetidamente hasta que una condición sea verdadera.

```

<instruccion-do-while> ::= "spawnner" <bloque> "exhausted"
                           <expresion-booleana> ";"
<instruccion-do-while> ::= "spawnner" <instruccion-simple> ";"
                           "exhausted" <expresion-booleana> ";"

```

### Instrucción For (walk VAR set <exp> to <exp> step <exp> craft <instrucción>)

La instrucción **for** en Notch Engine se representa mediante la palabra clave **walk**, evocando la idea de recorrer o caminar por un camino. Esta estructura permite iterar un bloque de código un número determinado de veces, con un contador que se incrementa (o decrementa) en cada iteración.

```

<instruccion-for> ::= "walk" <identificador> "set" <expresion> "to"
                      <expresion> "craft" <bloque>
<instruccion-for> ::= "walk" <identificador> "set" <expresion> "to"
                      <expresion> "step" <expresion> "craft" <bloque>
<instruccion-for> ::= "walk" <identificador> "set" <expresion> "to"
                      <expresion> "craft" <instruccion-simple> ";"
<instruccion-for> ::= "walk" <identificador> "set" <expresion> "to"
                      <expresion> "step" <expresion> "craft"
                      <instruccion-simple> ";"

```

### Instrucción With (withr <Referencia a Record> craft <instrucción>)

La instrucción **with** en Notch Engine se representa mediante la palabra clave **withr**, haciendo un juego de palabras con "withz" refiriéndose al enemigo Wither de Minecraft. Esta estructura permite trabajar con los campos de un registro sin tener que usar el operador de acceso repetidamente.

```

<instruccion-with> ::= "wither" <identificador> "craft" <bloque>
<instruccion-with> ::= "wither" <identificador> "craft"
                        <instruccion-simple> ";"

```

### Instrucción break (creeper)

La instrucción **break** en Notch Engine se representa mediante la palabra clave **creeper**, evocando cómo los creepers en Minecraft interrumpen abruptamente lo que el jugador está haciendo. Esta instrucción permite salir prematuramente de un bucle.

```

<instruccion-break> ::= "creeper" ";"

```

### Instrucción continue (enderPearl)

La instrucción **continue** en Notch Engine se representa mediante la palabra clave **enderPearl**, evocando cómo las Ender Pearls en Minecraft permiten teletransportarse, similar a cómo la instrucción **continue** "salta" por encima del resto del código del bucle. Esta instrucción permite saltar a la siguiente iteración de un bucle.

```

<instruccion-continue> ::= "enderPearl" ";"

```

### Instrucción Halt (ragequit)

La instrucción **halt** en Notch Engine se representa mediante la palabra clave **ragequit**, evocando la idea de abandonar abruptamente el juego por frustración o enojo. Esta instrucción termina inmediatamente la ejecución del programa.

```

<instruccion-halt> ::= "ragequit" ";"
<instruccion-halt> ::= "ragequit" "(" <expresion-entera> ")" ";"

```

### Ejemplos completos de estructuras de control

A continuación se presentan ejemplos que demuestran el uso de las diferentes estructuras de control en un programa Notch Engine:

```

WorldName EstructurasControl:

```

```

SpawnPoint

```

```

PolloCrudo

```

```

    $* Bloque simple *$

```

```

PolloCrudo
    Stack contador = 0;
    Stack suma = 0;
    soulsand contador;
PolloAsado

$* Instrucción while (repeater) *$
contador = 1;
repeater contador <= 5 craft
    PolloCrudo
        dropperSpider("Iteración: " bind contador);
        suma += contador;
        soulsand contador;
    PolloAsado

dropperSpider("Suma total: " bind suma);

$* Instrucción if-then-else (target) *$
Stack edad = 18;

target edad >= 18 craft hit
    PolloCrudo
        dropperSpider("Mayor de edad");
    PolloAsado
miss
    PolloCrudo
        dropperSpider("Menor de edad");
    PolloAsado

$* Instrucción switch (jukebox) *$
Stack opcion = 2;

jukebox opcion craft
    disc 1:
        PolloCrudo
            dropperSpider("Opción 1 seleccionada");
        PolloAsado

    disc 2:
        PolloCrudo
            dropperSpider("Opción 2 seleccionada");

```

```

    PolloAsado

    silence:
    PolloCrudo
        dropperSpider("Opción no reconocida");
    PolloAsado

$* Instrucción do-while (spawner-exhausted) *$
Stack intentos = 0;
Torch exito = Off;

spawner
    PolloCrudo
        soulsand intentos;
        dropperSpider("Intento número: " bind intentos);

        target intentos is 3 craft hit
            PolloCrudo
                exito = On;
            PolloAsado
    PolloAsado
exhausted
    exito;

$* Instrucción for (walk) *$
Stack total = 0;

walk i set 1 to 10 craft
PolloCrudo
    target i % 2 is 0 craft hit
        PolloCrudo
            total += i;
        PolloAsado
PolloAsado

dropperSpider("Suma de pares: " bind total);

$* Instrucción with (wither) *$
Entity Punto
PolloCrudo

```

```

    Stack x;
    Stack y;
    Stack z;
PolloAsado;

Entity Punto origen;

with origin craft
    PolloCrudo
        x = 0;
        y = 0;
        z = 0;
        dropperSpider("Punto en origen: (" bind x bind ",
            " bind y bind ", " bind z bind ")");
    PolloAsado

*$ Instrucciones break y continue *$
Stack i = 0;

repeater i < 10 craft
    PolloCrudo
        soulsand i;

        target i is 5 craft hit
            PolloCrudo
                dropperSpider("Saltando iteración 5");
                enderPearl;
            PolloAsado

        target i is 8 craft hit
            PolloCrudo
                dropperSpider("Terminando bucle en iteración 8");
                creeper;
            PolloAsado

        dropperSpider("Procesando: " bind i);
    PolloAsado

*$ Instrucción halt (ragequit) *$
target sum(1, 2) isNot 3 craft hit
    PolloCrudo

```

```

        dropperSpider("Error crítico: ¡Las matemáticas no funcionan!");
        ragequit;
    PolloAsado

    dropperSpider("Programa completado con éxito");
PolloAsado

worldSave

```

## 1.8. Funciones y Procedimientos - Gramática BNF

Esta sección define la gramática BNF para las funciones y procedimientos en Notch Engine, incluyendo su declaración, parámetros y retorno de valores.

### Encabezado de funciones (Spell <id>(<parameters>) – > <tipo>)

Las funciones en Notch Engine se declaran mediante la palabra clave **Spell**, evocando la idea de un hechizo o encantamiento en Minecraft. Una función procesa parámetros de entrada y devuelve un valor del tipo especificado.

```

<funcion> ::= "Spell" <identificador> "(" <lista-parametros> ")"
           "->" <tipo> <bloque>
<funcion> ::= "Spell" <identificador> "(" " )" "->" <tipo> <bloque>

<llamada-funcion> ::= <identificador> "(" <lista-argumentos> ")"
<llamada-funcion> ::= <identificador> "(" " )"

<lista-argumentos> ::= <expresion> "," <lista-argumentos>
<lista-argumentos> ::= <expresion>

```

### Encabezado de procedimientos (Ritual <id>(<parameters>))

Los procedimientos en Notch Engine se declaran mediante la palabra clave **Ritual**, evocando la idea de una ceremonia o ritual en Minecraft. A diferencia de las funciones, los procedimientos no devuelven un valor explícito; se utilizan principalmente por sus efectos secundarios.

```

<procedimiento> ::= "Ritual" <identificador> "(" <lista-parametros> ")"
                  <bloque>

```



```

<procedimiento> ::= "Ritual" <identificador> "(" ")" <bloque>

<llamada-procedimiento> ::= "ender_pearl" <identificador> "("
    <lista-argumentos> ")" ";"
<llamada-procedimiento> ::= "ender_pearl" <identificador> "(" ")" ";"

```

**Manejo de parámetros formales (( <type> :: <name>, <name>;  
<type> ref <name>; ... ))**

Notch Engine utiliza una sintaxis especial para declarar parámetros en funciones y procedimientos. Los parámetros se agrupan por tipo, separados por comas si son del mismo tipo y por punto y coma si son de diferentes tipos. La palabra clave `ref` indica que un parámetro se pasa por referencia.

```

<lista-parametros> ::= <grupo-parametros> ";" <lista-parametros>
<lista-parametros> ::= <grupo-parametros>

<grupo-parametros> ::= <tipo> "::" <lista-nombres-parametros>
<grupo-parametros> ::= <tipo> "ref" <identificador>

<lista-nombres-parametros> ::= <identificador> ","
    <lista-nombres-parametros>
<lista-nombres-parametros> ::= <identificador>

```

**Manejo de parámetros reales ((5,A,4,B))**

Los parámetros reales son las expresiones o valores que se pasan a una función o procedimiento cuando se realiza una llamada. En Notch Engine, se pasan entre paréntesis y separados por comas.

```

<llamada-funcion> ::= <identificador> "(" <lista-argumentos> ")"
<llamada-funcion> ::= <identificador> "(" ")"

<llamada-procedimiento> ::= "ender_pearl" <identificador> "("
    <lista-argumentos> ")" ";"
<llamada-procedimiento> ::= "ender_pearl" <identificador> "(" ")" ";"

<lista-argumentos> ::= <expresion> "," <lista-argumentos>
<lista-argumentos> ::= <expresion>

```

## Instrucción return (respawn)

La instrucción `return` en Notch Engine se representa mediante la palabra clave `respawn`, evocando la mecánica de reaparecer.<sup>en</sup> Minecraft. Esta instrucción permite devolver un valor desde una función o finalizar la ejecución de un procedimiento antes de tiempo.

```
<instruccion-return> ::= "respawn" <expresion> ";"  
<instruccion-return> ::= "respawn" ";"
```

## Ejemplos completos de funciones y procedimientos

A continuación se presentan ejemplos que demuestran el uso de funciones y procedimientos en un programa Notch Engine:

WorldName FuncionesProcedimientos:

CraftingTable

```
$* Función simple sin parámetros *$
```

```
Spell obtenerVersion() -> Spider
```

```
PolloCrudo
```

```
    respawn "Notch Engine v1.0";
```

```
PolloAsado
```

```
$* Función con parámetros *$
```

```
Spell calcularArea(Stack :: base, altura) -> Stack
```

```
PolloCrudo
```

```
    Stack area = base * altura // 2;
```

```
    respawn area;
```

```
PolloAsado
```

```
$* Función con parámetros de diferentes tipos *$
```

```
Spell verificarEdad(Stack :: edad; Spider nombre) -> Torch
```

```
PolloCrudo
```

```
    target edad >= 18 craft hit PolloCrudo
```

```
        dropperSpider(nombre bind " es mayor de edad");
```

```
        respawn On;
```

```
    PolloAsado miss PolloCrudo
```

```
        dropperSpider(nombre bind " es menor de edad");
```

```
        respawn Off;
```

```
    PolloAsado
```

```
PolloAsado
```

```

*$ Función con parámetro por referencia *$
Spell incrementar(Stack ref contador) -> Stack
PolloCrudo
    soulsand contador;
    respawn contador;
PolloAsado

*$ Procedimiento simple sin parámetros *$
Ritual mostrarMensajeBienvenida()
PolloCrudo
    dropperSpider("Bienvenido a Notch Engine");
    dropperSpider("=====");
PolloAsado

*$ Procedimiento con parámetros *$
Ritual mostrarInformacion(Spider :: nombre; Stack nivel, salud)
PolloCrudo
    dropperSpider("Información del jugador:");
    dropperSpider("Nombre: " bind nombre);
    dropperSpider("Nivel: " bind nivel);
    dropperSpider("Salud: " bind salud);
PolloAsado

*$ Procedimiento con retorno anticipado *$
Ritual procesarDatos(Stack :: valor)
PolloCrudo
    target valor <= 0 craft hit PolloCrudo
    dropperSpider("Error: El valor debe ser positivo");
    respawn;  $* Retorno anticipado sin valor *$
PolloAsado

    dropperSpider("Procesando valor: " bind valor);
    $* Resto del procesamiento... *$
PolloAsado

*$ Función recursiva *$
Spell factorial(Stack :: n) -> Stack
PolloCrudo
    target n <= 1 craft hit PolloCrudo
    respawn 1;

```

```

    PolloAsado

    Stack resultado = n * factorial(n - 1);
    respawn resultado;
PolloAsado

SpawnPoint
PolloCrudo
    $* Llamadas a funciones *$
    Spider version = obtenerVersion();
    dropperSpider("Versión: " bind version);

    Stack area = calcularArea(5, 8);
    dropperSpider("Área del triángulo: " bind area);

    Torch esMayor = verificarEdad(20, "Juan");

    $* Llamada a función con parámetro por referencia *$
    Stack contador = 5;
    Stack nuevoValor = incrementar(contador);
    dropperSpider("Contador: " bind contador);
    dropperSpider("Valor retornado: " bind nuevoValor);

    $* Llamadas a procedimientos *$
    ender_pearl mostrarMensajeBienvenida();

    ender_pearl mostrarInformacion("Alex", 10, 18);

    ender_pearl procesarDatos(0); $* Este mostrará error y terminará *$
    ender_pearl procesarDatos(5); $* Este procesará normalmente *$

    $* Llamada a función recursiva *$
    Stack resultado = factorial(5);
    dropperSpider("Factorial de 5: " bind resultado);
PolloAsado

worldSave

```

## 1.9. Elementos Auxiliares - Gramática BNF

Esta sección define la gramática BNF para los elementos auxiliares en Notch Engine, incluyendo operaciones especiales, entrada/salida estándar, y elementos sintácticos fundamentales.

### Operación de size of (chunk <exp> o <tipo>)

La operación **chunk** en Notch Engine permite determinar el tamaño en bytes de una variable o tipo de dato. Esta operación recibe una expresión o un tipo y devuelve un valor entero.

```
<operacion-tamano> ::= "chunk" <expresion>
<operacion-tamano> ::= "chunk" <tipo>
```

### Sistema de coerción de tipos (<exp> >> <tipo>)

El sistema de coerción de tipos en Notch Engine, representado por el operador >>, permite interpretar el resultado de una expresión como si fuera de otro tipo sin convertirlo completamente.

```
<cohercion-tipo> ::= <expresion> ">>" <tipo>
```

### Manejo de la entrada estándar (x = hopper<TipoBásico>())

El manejo de la entrada estándar en Notch Engine se realiza mediante funciones predefinidas que comienzan con la palabra **hopper**, seguida del tipo de dato que se espera leer.

```
<entrada-estandar> ::= "hopperStack" "(" ")"
<entrada-estandar> ::= "hopperRune" "(" ")"
<entrada-estandar> ::= "hopperSpider" "(" ")"
<entrada-estandar> ::= "hopperTorch" "(" ")"
<entrada-estandar> ::= "hopperChest" "(" ")"
<entrada-estandar> ::= "hopperGhast" "(" ")"
```

```
<instruccion-entrada> ::= <identificador> "=" <entrada-estandar> ";"
```

### Manejo de la salida estándar (dropper<tipoBásico>(dato))

El manejo de la salida estándar en Notch Engine se realiza mediante funciones predefinidas que comienzan con la palabra **dropper**, seguida del tipo de dato que se desea mostrar.

```

<salida-estandar> ::= "dropperStack" "(" <expresion> ")"
<salida-estandar> ::= "dropperRune" "(" <expresion> ")"
<salida-estandar> ::= "dropperSpider" "(" <expresion> ")"
<salida-estandar> ::= "dropperTorch" "(" <expresion> ")"
<salida-estandar> ::= "dropperChest" "(" <expresion> ")"
<salida-estandar> ::= "dropperGhast" "(" <expresion> ")"

```

```

<instruccion-salida> ::= <salida-estandar> ";"

```

### Terminador o separador de instrucciones - Instrucción nula (;)

El punto y coma (;) se utiliza en Notch Engine como terminador de instrucciones simples. También puede aparecer solo para representar una instrucción nula o vacía.

```

<instruccion-simple> ::= <asignacion> ";"
<instruccion-simple> ::= <expresion-incremento> ";"
<instruccion-simple> ::= <llamada-procedimiento> ";"
<instruccion-simple> ::= <operacion-conjunto> ";"
<instruccion-simple> ::= <operacion-archivo> ";"
<instruccion-simple> ::= <instruccion-entrada> ";"
<instruccion-simple> ::= <instruccion-salida> ";"
<instruccion-simple> ::= ";"

```

### Todo programa se debe cerrar con un (worldSave)

Cada programa en Notch Engine debe terminar con la palabra clave worldSave, que simboliza la acción de guardar el mundo en Minecraft.

```

<programa> ::= <titulo-programa> <secciones> <punto-entrada> "worldSave"

```

### Expresiones generales

Las expresiones en Notch Engine pueden ser desde simples literales o identificadores hasta combinaciones complejas de operadores y llamadas a funciones.

```

<expresion> ::= <literal>
<expresion> ::= <identificador>
<expresion> ::= <expresion-aritmetica-enteros>
<expresion> ::= <expresion-aritmetica-flotante>
<expresion> ::= <expresion-logica>

```



```

<secciones> ::= <seccion-constantes> <seccion-prototipos>
<secciones> ::= <seccion-constantes> <seccion-rutinas>
<secciones> ::= <seccion-tipos> <seccion-variables>
<secciones> ::= <seccion-tipos> <seccion-prototipos>
<secciones> ::= <seccion-tipos> <seccion-rutinas>
<secciones> ::= <seccion-variables> <seccion-prototipos>
<secciones> ::= <seccion-variables> <seccion-rutinas>
<secciones> ::= <seccion-prototipos> <seccion-rutinas>
<secciones> ::= <seccion-constantes>
<secciones> ::= <seccion-tipos>
<secciones> ::= <seccion-variables>
<secciones> ::= <seccion-prototipos>
<secciones> ::= <seccion-rutinas>
<secciones> ::=

```

```

<punto-entrada> ::= "SpawnPoint" <bloque>

```

## Ejemplos completos de elementos auxiliares

A continuación se presentan ejemplos que demuestran el uso de los diferentes elementos auxiliares en un programa Notch Engine:

WorldName ElementosAuxiliares:

Bedrock

```

Obsidian Stack MAX_BUFFER_SIZE 1024;
Obsidian Spider VERSION "1.0";

```

Inventory

```

$* Variables para los ejemplos *$
Stack contador;
Spider texto = "Notch Engine";
Shelf[5] numeros;

```

Entity Punto

PolloCrudo

```

Stack x;

```

```

Stack y;

```

PolloAsado;



Entity Punto origen;

SpawnPoint

```
$* Operación de tamaño (chunk) *$
Stack tamanoStack = chunk Stack;
Stack tamanoSpider = chunk Spider;
Stack tamanoArreglo = chunk numeros;

dropperSpider("Tamaño de un Stack en bytes: " bind tamanoStack);
dropperSpider("Tamaño de un Spider en bytes: " bind tamanoSpider);
dropperSpider("Tamaño del arreglo en bytes: " bind tamanoArreglo);

$* Sistema de coerción de tipos (>>) *$
Ghast decimal = 3.75;
Stack entero = decimal >> Stack; $* entero = 3 (truncado) *$

dropperSpider("Valor original: " bind decimal);
dropperSpider("Valor coercionado: " bind entero);

$* Manejo de la entrada estándar (hopper) *$
dropperSpider("Ingrese un número entero:");
contador = hopperStack();

dropperSpider("Ingrese su nombre:");
Spider nombre = hopperSpider();

$* Manejo de la salida estándar (dropper) *$
dropperSpider("Hola, " bind nombre bind "!");
dropperStack(contador);
dropperGhast(3.14159);
dropperTorch(0n);

$* Terminador o separador de instrucciones - Instrucción nula (;) *$
contador = 0;
soulsand contador;
; $* Instrucción nula *$
soulsand contador;

$* Bloques de código con comentarios *$
PolloCrudo
```

```
$* Este es un comentario de bloque
    que puede abarcar múltiples líneas
    y es ignorado por el compilador *$

contador = 5;  $$ Este es un comentario de línea

target contador > 0 craft hit PolloCrudo
    dropperSpider("Contador es positivo");
    $$ Otro comentario de línea
PolloAsado
PolloAsado
```

worldSave

# Capítulo 2

## Documentacion del Parser

### 2.1. Documentacion inicial

### 2.2. Gramática del Parser

El parser del compilador Notch-Engine fue construido a partir de una gramática definida en formato BNF utilizando la herramienta **GikGram**. Esta herramienta facilita el diseño, validación y depuración de gramáticas LL(1), asegurando que la gramática sea adecuada para un compilador dirigido por tabla. GikGram también permite detectar errores comunes como recursividad por la izquierda, reglas no alcanzables o conflictos de predicción, lo cual fue fundamental para alcanzar una versión funcional del parser.

#### Ventajas de usar GikGram

- Permite documentar la gramática de forma estructurada en Excel.
- Verifica automáticamente la validez LL(1) y genera las tablas necesarias.
- Genera código compatible con C/C++ y Java, agilizando la integración.
- Identifica errores comunes como recursividad o ambigüedad.

#### Errores LL(1) que GikGram puede detectar

Durante el desarrollo, se corrigieron diversos problemas LL(1) con ayuda de GikGram, tales como:

1. Símbolos terminales o no terminales no definidos.

2. No terminales no alcanzables desde el símbolo inicial.
3. Reglas que no aterrizan (no terminan en terminales).
4. Necesidad de factorización para evitar conflictos de predicción.
5. Recursividad directa o indirecta por la izquierda.
6. Doble predicción (conflictos de parsing en una misma celda).

## Estructura general de la gramática

La gramática se encuentra estructurada por secciones que reflejan directamente la organización del lenguaje Notch-Engine:

- **Definición inicial:** Define el punto de entrada del programa ('World-Name') y su clausura ('WorldSave').
- **Constantes y Tipos:** Utiliza 'Obsidian' para declarar constantes, y 'Anvil' para asociar identificadores con tipos.
- **Inventario:** Se declaran variables mediante tipos y listas de identificadores, permitiendo inicialización.
- **Recetas:** Define prototipos de funciones ('Spell') y procedimientos ('Ritual').
- **Funciones:** Define implementaciones de funciones y procedimientos, incluyendo bloques de código.
- **SpawnPoint:** Secciones con instrucciones ('Statements') que representan el cuerpo ejecutable del programa.
- **Control de flujo:** Instrucciones como 'target', 'repeter', 'walk', entre otras.
- **Expresiones:** Soporta expresiones lógicas, aritméticas, flotantes, coerción y llamadas a funciones.
- **Asignaciones y Accesos:** Define operadores de asignación y accesos a registros, arreglos o campos.
- **Tipos y Literales:** Se listan los tipos primitivos ('Stack', 'Spider', etc.) y las formas válidas de literal (números, cadenas, booleanos, registros).

## Formato de la gramática

Se usó una notación BNF simplificada, donde:

- Los no terminales están entre ángulos ‘ $\langle \rangle$ ’.
- Se usa ‘ $::=$ ’ para indicar producción.
- Las reglas están escritas una por línea.
- Las producciones epsilon están representadas por reglas vacías.
- Los símbolos semánticos se indican con ‘ $\#$ ’ (por ejemplo, ‘ $\text{init}_{tsg}$ ’).  
A continuación, se presenta la gramática completa organizada por secciones, tal como fue integrada en el archivo de entrada de GikGram y utilizada por el parser LL(1) generado.

### Definición inicial

Esta sección define la estructura general del programa. Todo programa debe comenzar con un nombre (**WorldName**) y finalizar con la palabra clave **WorldSave**. Entre ambos, se incluyen las distintas secciones lógicas del lenguaje: constantes, tipos, variables, prototipos, implementaciones y sentencias. El uso de símbolos semánticos (**#init\_tsg**, **#free\_tsg**) permite inicializar y liberar la tabla de símbolos global.

```
<program>          ::= #init_tsg WORLD_NAME IDENTIFICADOR DOS_PUNTOS
                    <program_sections> WORLD_SAVE #free_tsg
<program_sections> ::= <section_list>
<section_list>     ::= <section> <section_list>
<section_list>     ::=
<section>          ::= BEDROCK <bedrock_section>
<section>          ::= RESOURCE_PACK <resource_pack_section>
<section>          ::= INVENTORY <inventory_section>
<section>          ::= RECIPE <recipe_section>
<section>          ::= CRAFTING_TABLE <crafting_table_section>
<section>          ::= SPAWN_POINT <spawn_point_section>
<bedrock_section> ::= <constant_decl> <bedrock_section>
<bedrock_section> ::=
```

### Variables & Constantes

Esta sección agrupa las producciones responsables de declarar constantes, tipos definidos por el usuario y variables. Las constantes se definen en la

sección **Bedrock**, los tipos en **ResourcePack**, y las variables en **Inventory**. Se permite la inicialización de variables y la definición de múltiples identificadores en una sola línea. También se incluye el inicio de la sección de recetas con prototipos de funciones.

```

<constant_decl>      ::= OBSIDIAN <type> IDENTIFICADOR <value> PUNTO_Y_COMA
<value>              ::= <literal>
<value>              ::=
<resource_pack_section> ::= <type_decl> <resource_pack_section>
<resource_pack_section> ::=
<type_decl>          ::= ANVIL IDENTIFICADOR FLECHA <type> PUNTO_Y_COMA
<inventory_section> ::= <var_decl> <inventory_section>
<inventory_section> ::=
<var_decl>           ::= <type> <var_list> PUNTO_Y_COMA
<var_list>           ::= IDENTIFICADOR <var_init> <more_vars>
<var_init>           ::= IGUAL <expression>
<var_init>           ::=
<more_vars>          ::= COMA IDENTIFICADOR <var_init> <more_vars>
<more_vars>          ::=
<recipe_section>     ::= <prototype> <recipe_section>
<recipe_section>     ::=

```

## Funciones

Esta parte de la gramática define tanto los prototipos como las implementaciones de funciones (**Spell**) y procedimientos (**Ritual**). Los prototipos se declaran en la sección **Recipe** y las implementaciones en la sección **CraftingTable**. Las funciones incluyen tipo de retorno y bloque de código, mientras que los procedimientos no retornan valor. Se utilizan símbolos semánticos para validar la estructura y el retorno de funciones.

```

<prototype>          ::= SPELL IDENTIFICADOR PARENTESIS_ABRE <params> PARENTE
<prototype>          ::= RITUAL <proc_prototype_tail>
<proc_prototype_tail> ::= IDENTIFICADOR PARENTESIS_ABRE <params> PARENTESIS_CI
<crafting_table_section> ::= <function> <crafting_table_section>
<crafting_table_section> ::=
<function>           ::= SPELL <func_impl_tail>
<function>           ::= RITUAL <proc_impl_tail>
<func_impl_tail>     ::= #chk_func_start IDENTIFICADOR PARENTESIS_ABRE <param
<proc_impl_tail>     ::= IDENTIFICADOR PARENTESIS_ABRE <params> PARENTESIS_CI
<spawn_point_section> ::= <statement> <spawn_point_section>
<spawn_point_section> ::=

```

## Statements

Esta sección agrupa todas las instrucciones válidas dentro de bloques de código. Incluye sentencias simples como `RAGEQUIT`, llamadas a funciones, asignaciones y estructuras de control como `if`, `while`, `switch` o `for`. También contempla bloques anidados delimitados por `PolloCrudo` y `PolloAsado`, así como instrucciones especiales relacionadas con el entorno del lenguaje.

```
<statement>      ::= RAGEQUIT PUNTO_Y_COMA
<statement>      ::= RESPAWN <expression> PUNTO_Y_COMA
<statement>      ::= MAKE PARENTESIS_ABRE <file_literal> PARENTESIS_CIERRA PUNTO_Y_COMA
<statement>      ::= <ident_stmt> PUNTO_Y_COMA
<ident_stmt>     ::= <assignment>
<ident_stmt>     ::= <func_call>
<statement>      ::= <if_stmt>
<statement>      ::= <while_stmt>
<statement>      ::= <repeat_stmt>
<statement>      ::= <for_stmt>
<statement>      ::= <switch_stmt>
<statement>      ::= <with_stmt>
<statement>      ::= CREEPER PUNTO_Y_COMA
<statement>      ::= ENDER_PEARL PUNTO_Y_COMA
<statement>      ::= RETURN <return_expr> PUNTO_Y_COMA
<statement>      ::= <block>
<block>          ::= POLLO_CRUDO <statements> POLLO_ASADO
<statements>     ::= <statement> <statements>
<statements>     ::=
<statement>      ::= UNLOCK PARENTESIS_ABRE <expression> PARENTESIS_CIERRA PUNTO_Y_COMA
<statement>      ::= LOCK PARENTESIS_ABRE <expression> PARENTESIS_CIERRA PUNTO_Y_COMA
<statement>      ::= FORGE PARENTESIS_ABRE <expression> PARENTESIS_CIERRA PUNTO_Y_COMA
<statement>      ::= GATHER PARENTESIS_ABRE <expression> PARENTESIS_CIERRA PUNTO_Y_COMA
<statement>      ::= TAG PARENTESIS_ABRE <expression> PARENTESIS_CIERRA PUNTO_Y_COMA
<statement>      ::= SOULSAND IDENTIFICADOR PUNTO_Y_COMA
<statement>      ::= MAGMA IDENTIFICADOR PUNTO_Y_COMA
<expression>     ::= CHUNK PARENTESIS_ABRE <expression> PARENTESIS_CIERRA
```

## Control

Esta sección define las estructuras de control del lenguaje, como condicionales, ciclos y estructuras de selección. Las palabras clave están inspiradas en la temática del lenguaje, por ejemplo, `TARGET` para `if`, `REPEATER` para

while, y JUKEBOX para switch. Se incluyen símbolos semánticos para validar condiciones como múltiples default en los casos.

```

<if_stmt>          ::= TARGET <expression> CRAFT HIT <statement> <else_part>
<else_part>       ::= MISS <statement>
<while_stmt>      ::= REPEATER <expression> CRAFT <statement>
<repeat_stmt>     ::= SPAWNER <statement> EXHAUSTED <expression> PUNTO_Y_COMA
<for_stmt>        ::= WALK IDENTIFICADOR SET <expression> TO <expression> <step_p
<step_part>       ::= STEP <expression>
<step_part>       ::=
<switch_stmt>     ::= #sw1 JUKEBOX <expression> CRAFT <case_list> <default_case>
<case_list>       ::= DISC <literal> DOS_PUNTOS <statement> <case_list>
<case_list>       ::=
<default_case>    ::= #sw2 SILENCE DOS_PUNTOS <statement>
<with_stmt>       ::= WITHER IDENTIFICADOR CRAFT <statement>

```

## Asignación

Esta parte de la gramática define las expresiones de asignación. Se permite modificar valores mediante distintos operadores de asignación estándar y extendidos, incluyendo variantes para operaciones con números flotantes. La parte izquierda de la asignación es un acceso a memoria y la derecha una expresión evaluable.

```

<assignment>      ::= <access_path> <assign_op> <expression>
<assign_op>       ::= IGUAL
<assign_op>       ::= SUMA_FLOTANTE_IGUAL
<assign_op>       ::= RESTA_FLOTANTE_IGUAL
<assign_op>       ::= MULTIPLICACION_FLOTANTE_IGUAL
<assign_op>       ::= DIVISION_FLOTANTE_IGUAL
<assign_op>       ::= MODULO_FLOTANTE_IGUAL
<assign_op>       ::= SUMA_IGUAL
<assign_op>       ::= RESTA_IGUAL
<assign_op>       ::= MULTIPLICACION_IGUAL
<assign_op>       ::= DIVISION_IGUAL
<assign_op>       ::= MODULO_IGUAL

```

## Expresiones

Las expresiones en Notch-Engine pueden ser de tipo especial, numérico, lógico, relacional, aritmético o flotante. Esta sección permite combinar



y evaluar expresiones mediante operadores definidos por el lenguaje, incluyendo funciones especiales como `HASH`, `BIND` o `SEEK`. También se contemplan expresiones compuestas mediante operadores lógicos y relacionales.

```

<expression>      ::= <special_expr>
<special_expr>    ::= ETCH_UP PARENTESIS_ABRE <expression> PARENTESIS_CIERRA
<special_expr>    ::= ETCH_DOWN PARENTESIS_ABRE <expression> PARENTESIS_CIERRA
<special_expr>    ::= IS_ENGRAVED PARENTESIS_ABRE <expression> PARENTESIS_CIERRA
<special_expr>    ::= IS_INSCRIBED PARENTESIS_ABRE <expression> PARENTESIS_CIERRA
<special_expr>    ::= HASH PARENTESIS_ABRE <expression> PARENTESIS_CIERRA
<special_expr>    ::= BIND PARENTESIS_ABRE <expression> PARENTESIS_CIERRA
<special_expr>    ::= FROM PARENTESIS_ABRE <expression> PARENTESIS_CIERRA
<special_expr>    ::= EXCEPT PARENTESIS_ABRE <expression> PARENTESIS_CIERRA
<special_expr>    ::= SEEK PARENTESIS_ABRE <expression> PARENTESIS_CIERRA
<special_expr>    ::= ADD PARENTESIS_ABRE <expression> PARENTESIS_CIERRA
<special_expr>    ::= DROP PARENTESIS_ABRE <expression> PARENTESIS_CIERRA
<special_expr>    ::= FEED PARENTESIS_ABRE <expression> PARENTESIS_CIERRA
<special_expr>    ::= MAP PARENTESIS_ABRE <expression> PARENTESIS_CIERRA
<special_expr>    ::= BIOM PARENTESIS_ABRE <expression> PARENTESIS_CIERRA
<special_expr>    ::= VOID PARENTESIS_ABRE <expression> PARENTESIS_CIERRA
<special_expr>    ::= <numeric_expr>
<numeric_expr>    ::= <common_expr>
<common_expr>    ::= <logical_expr>
<common_expr>    ::= <arithmetic_expr>
<common_expr>    ::= <float_expr>
<logical_expr>    ::= <relational_expr> <logical_tail>
<logical_tail>    ::= XOR <relational_expr> <logical_tail>
<logical_tail>    ::= AND <relational_expr> <logical_tail>
<logical_tail>    ::= OR <relational_expr> <logical_tail>
<logical_tail>    ::=
<relational_expr> ::= <arithmetic_expr> <relational_tail>
<relational_tail> ::= <rel_op> <arithmetic_expr> <relational_tail>
<relational_tail> ::=

```

## Relaciones

Esta sección define los operadores relacionales que permiten comparar expresiones. Son utilizados en expresiones condicionales y estructuras de control para evaluar igualdad, desigualdad y orden.

```
<rel_op> ::= DOBLE_IGUAL
```

```

<rel_op> ::= MENOR_QUE
<rel_op> ::= MAYOR_QUE
<rel_op> ::= MENOR_IGUAL
<rel_op> ::= MAYOR_IGUAL
<rel_op> ::= IS
<rel_op> ::= IS_NOT

```

## Aritmética

Esta sección define expresiones aritméticas y flotantes. Se permite el uso de operaciones básicas como suma, resta, multiplicación, división y módulo, en versiones enteras y flotantes. También se consideran factores con coerción de tipos, uso de paréntesis, llamados a funciones y operadores unarios.

```

<arithmetic_expr>      ::= <term> <arithmetic_tail>
<arithmetic_tail>     ::= SUMA <term> <arithmetic_tail>
<arithmetic_tail>     ::= RESTA <term> <arithmetic_tail>
<arithmetic_tail>     ::=
<term>                 ::= <factor> <term_tail>
<term_tail>           ::= MULTIPLICACION <factor> <term_tail>
<term_tail>           ::= DIVISION <factor> <term_tail>
<term_tail>           ::= MODULO <factor> <term_tail>
<term_tail>           ::=
<float_expr>          ::= <float_term> <float_tail>
<float_tail>          ::= SUMA_FLOTANTE <float_term> <float_tail>
<float_tail>          ::= RESTA_FLOTANTE <float_term> <float_tail>
<float_tail>          ::=
<float_term>          ::= <float_factor> <float_term_tail>
<float_term_tail>     ::= MULTIPLICACION_FLOTANTE <float_factor> <float_term_tail>
<float_term_tail>     ::= DIVISION_FLOTANTE <float_factor> <float_term_tail>
<float_term_tail>     ::= MODULO_FLOTANTE <float_factor> <float_term_tail>
<float_term_tail>     ::=
<float_factor>        ::= PARENTESIS_ABRE <float_expr> PARENTESIS_CIERRA
<float_factor>        ::= NUMERO_DECIMAL
<float_factor>        ::= <id_expr>
<id_expr>             ::= <access_path>
<id_expr>             ::= <func_call>
<factor>              ::= <unary_op> <factor>
<factor>              ::= <primary> <coercion_tail>
<coercion_tail>       ::= COERCION <type> <coercion_tail>
<coercion_tail>       ::=

```

```

<primary>          ::= PARENTESIS_ABRE <expression> PARENTESIS_CIERRA
<primary>          ::= <literal>
<primary>          ::= <id_expr>
<unary_op>         ::= SUMA
<unary_op>         ::= RESTA
<unary_op>         ::= NOT

```

## Funciones

Esta sección define la sintaxis para las llamadas a funciones, tanto genéricas como especiales del lenguaje, como las variantes **DROPPER** y **HOPPER**. Todas las llamadas utilizan paréntesis y pueden recibir argumentos separados por comas. Estas funciones se pueden usar dentro de expresiones.

```

<func_call> ::= IDENTIFICADOR PARENTESIS_ABRE <args> PARENTESIS_CIERRA
<func_call> ::= DROPPER_STACK PARENTESIS_ABRE <expression> PARENTESIS_CIERRA
<func_call> ::= DROPPER_RUNE PARENTESIS_ABRE <expression> PARENTESIS_CIERRA
<func_call> ::= DROPPER_SPIDER PARENTESIS_ABRE <expression> PARENTESIS_CIERRA
<func_call> ::= DROPPER_TORCH PARENTESIS_ABRE <expression> PARENTESIS_CIERRA
<func_call> ::= DROPPER_CHEST PARENTESIS_ABRE <expression> PARENTESIS_CIERRA
<func_call> ::= DROPPER_GHAST PARENTESIS_ABRE <expression> PARENTESIS_CIERRA
<func_call> ::= HOPPER_STACK PARENTESIS_ABRE <expression> PARENTESIS_CIERRA
<func_call> ::= HOPPER_RUNE PARENTESIS_ABRE <expression> PARENTESIS_CIERRA
<func_call> ::= HOPPER_SPIDER PARENTESIS_ABRE <expression> PARENTESIS_CIERRA
<func_call> ::= HOPPER_TORCH PARENTESIS_ABRE <expression> PARENTESIS_CIERRA
<func_call> ::= HOPPER_CHEST PARENTESIS_ABRE <expression> PARENTESIS_CIERRA
<func_call> ::= HOPPER_GHAST PARENTESIS_ABRE <expression> PARENTESIS_CIERRA
<args>       ::= <expression> <more_args>
<more_args>  ::= COMA <expression> <more_args>
<more_args>  ::=

```

## Acceso

Esta sección define cómo se accede a variables, campos de registros, posiciones de arreglos y elementos anidados. También incluye la estructura para la definición de parámetros formales en funciones o procedimientos, utilizando el operador `::.`

```

<access_path> ::= IDENTIFICADOR <access_tail>
<access_tail> ::= ARROBA IDENTIFICADOR <access_tail>
<access_tail> ::= CORCHETE_ABRE <expression> CORCHETE_CIERRA <access_tail>
<access_tail> ::= PUNTO IDENTIFICADOR <access_tail>

```

```

<access_tail> ::=
<params> ::= <param_group> <more_params>
<more_params> ::= COMA <param_group> <more_params>
<more_params> ::=
<param_group> ::= <type> DOS_PUNTOS DOS_PUNTOS <param_list>
<param_list> ::= IDENTIFICADOR <more_param_ids>
<more_param_ids> ::= COMA IDENTIFICADOR <more_param_ids>
<more_param_ids> ::=

```

## Type

Define los tipos de datos válidos en el lenguaje. Se permite el uso de referencias mediante el operador REF, así como tipos primitivos como STACK, RUNE, ENTITY, entre otros inspirados en el mundo de Minecraft.

```

<type> ::= REF <type>
<type> ::= STACK
<type> ::= RUNE
<type> ::= SPIDER
<type> ::= TORCH
<type> ::= CHEST
<type> ::= BOOK
<type> ::= GHAST
<type> ::= SHELF
<type> ::= ENTITY

```

## Literal

Define las posibles formas de valores constantes en el lenguaje. Incluye literales primitivos (números, cadenas, booleanos) y estructuras más complejas como arreglos, registros y sets. También contempla literales especiales de archivo.

```

<literal> ::= CORCHETE_ABRE <array_elements> CORCHETE_CIERRA
<literal> ::= LLAVE_ABRE <literal_contenido>
<literal_contenido> ::= <record_fields> LLAVE_CIERRA
<literal_contenido> ::= <literal_llave>
<literal> ::= NUMERO_ENTERO
<literal> ::= NUMERO_DECIMAL
<literal> ::= CADENA
<literal> ::= CARACTER
<liteal> ::= ON

```

```

<literal>                ::= OFF
<literal_llave>          ::= DOS_PUNTOS <set_elements> DOS_PUNTOS LLAVE_CIERRA
<literal_llave>          ::= BARRA <file_literal> BARRA LLAVE_CIERRA

```

## Elementos

Define los componentes internos de estructuras de datos como conjuntos, arreglos, registros y literales de archivo. También incluye la producción para las expresiones de retorno, necesarias en funciones o procedimientos.

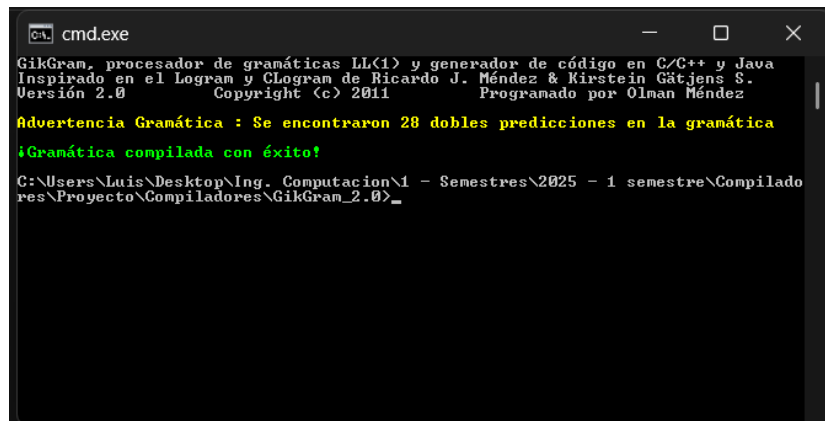
```

<set_elements>            ::= <expression> <more_set_elements>
<more_set_elements>      ::= COMA <expression> <more_set_elements>
<more_set_elements>      ::=
<file_literal>           ::= CADENA COMA CARACTER
<array_elements>         ::= <expression> <more_array_elements>
<more_array_elements>    ::= COMA <expression> <more_array_elements>
<more_array_elements>    ::=
<record_fields>          ::= IDENTIFICADOR DOS_PUNTOS <expression> <more_record_fie
<more_record_fields>     ::= COMA IDENTIFICADOR DOS_PUNTOS <expression> <more_recor
<more_record_fields>     ::=
<return_expr>            ::= <expression>
<return_expr>            ::= #check_is_procedure

```

### 2.2.1. Resultados de GikGram

En la Figura 2.1 se muestra un extracto de la validación LL(1) generada por GikGram. Esta validación ayudó a depurar múltiples errores como doble predicción, recursividad por la izquierda y símbolos no alcanzables, logrando así una versión funcional.



```
cmd.exe
GikGram, procesador de gramáticas LL(1) y generador de código en C/C++ y Java
Inspirado en el Logram y Clogram de Ricardo J. Méndez & Kirstein Gätjens S.
Versión 2.0 Copyright (c) 2011 Programado por Olman Méndez

Advertencia Gramática : Se encontraron 28 dobles predicciones en la gramática
¡Gramática compilada con éxito!

C:\Users\Luis\Desktop\Ing. Computacion\1 - Semestres\2025 - 1 semestre\Compiladores\Proyecto\Compiladores\GikGram_2.0>
```

Figura 2.1: Reporte de validación generado por GikGram

#### Documentación del código

Esta sección describe las principales funciones del parser implementado para el compilador Notch Engine, destacando su funcionalidad y propósito.

### 2.2.2. Clase Parser

- **(tokens, debug=False):** Constructor de la clase Parser. Inicializa el analizador sintáctico con una lista de tokens obtenida del scanner, eliminando comentarios y configurando opciones de depuración.
- **imprimir\_debug(mensaje, nivel=1):** Muestra mensajes de depuración según su nivel de importancia (1=crítico, 2=importante, 3=detallado), permitiendo controlar la cantidad de información mostrada durante el análisis.
- **imprimir\_estado\_pila(nivel=2):** Imprime una representación resumida del estado actual de la pila de análisis, mostrando los últimos 5 elementos para facilitar la depuración.
- **avanzar():** Avanza al siguiente token en la secuencia, manteniendo un historial de tokens procesados que facilita la recuperación de errores y el análisis contextual.
- **obtener\_tipo\_token():** Mapea el token actual al formato numérico esperado por la gramática, implementando casos especiales para identificadores como PolloCrudo, PolloAsado y worldSave.

- **match(*terminal\_esperado*):** Verifica si el token actual coincide con el terminal esperado, manejando casos especiales como identificadores que funcionan como palabras clave.
- **reportar\_error(*mensaje*):** Genera mensajes de error contextuales y específicos, incluyendo información sobre la ubicación del error y posibles soluciones.
- **sincronizar(*simbolo\_no\_terminal*):** Implementa la recuperación de errores avanzando hasta encontrar un token en el conjunto Follow del no terminal o un punto seguro predefinido.
- **obtener\_follows(*simbolo\_no\_terminal*):** Calcula el conjunto Follow para un no-terminal específico, fundamental para la recuperación de errores.
- **procesar\_no\_terminal(*simbolo\_no\_terminal*):** Procesa un símbolo no terminal aplicando la regla correspondiente según la tabla de parsing, incluyendo manejo de casos especiales.
- **parse():** Método principal que implementa el algoritmo de análisis sintáctico descendente predictivo, siguiendo el modelo de Driver de Parsing LL(1).
- **push(*simbolo*):** Añade un símbolo a la pila de análisis.
- **pop():** Elimina y retorna el símbolo superior de la pila de análisis.
- **sincronizar\_con\_follows(*simbolo*):** Sincroniza el parser usando el conjunto Follow del símbolo no terminal.
- **sincronizar\_con\_puntos\_seguros():** Sincroniza el parser usando puntos seguros predefinidos como delimitadores de bloques y sentencias.

### 2.2.3. Funciones auxiliares

- **parser(*tokens*, *debug=False*):** Función de conveniencia que crea una instancia del Parser y ejecuta el análisis sintáctico.
- **iniciar\_parser(*tokens*, *debug=False*, *nivel\_debug=3*):** Función principal para integrar el parser con el resto del compilador, configurando el nivel de detalle de la depuración y realizando un post-procesamiento de errores para suprimir falsos positivos.

### 2.2.4. Características relevantes

El parser implementa varias técnicas avanzadas, como:

- Recuperación eficiente de errores usando información contextual y conjuntos Follow
- Manejo de casos especiales para palabras clave que pueden aparecer como identificadores
- Filtrado inteligente de errores para evitar mensajes redundantes o falsos positivos
- Mecanismo de depuración multinivel para facilitar el diagnóstico durante el desarrollo
- Historial de tokens para mejorar el análisis contextual y la recuperación de errores

Esta implementación sigue fielmente el algoritmo de análisis sintáctico descendente recursivo con predicción basada en tablas LL(1), adaptado para manejar las particularidades del lenguaje Notch Engine.

Ademas se tienen metodos de apoyo y los generados por Gikgram, a continuacion se muestran:

### 2.2.5. SpecialTokens

**Descripción:** Clase que maneja casos especiales de tokens para el parser de Notch-Engine, especialmente útil para identificadores especiales como PolloCrudo/PolloAsado. **Métodos principales:**

- `handle_double_colon`: Maneja el token `::` simulando que son dos `DOS_PUNTOS`.
- `is_special_identifier`: Verifica si un token es un identificador especial.
- `get_special_token_type`: Obtiene el tipo especial correspondiente a un identificador.
- `get_special_token_code`: Obtiene el código numérico del token especial.
- `is_in_constant_declaration_context`: Determina si estamos en un contexto de declaración de constante.



- `is_literal_token`: Verifica si un tipo de token es un literal.
- `suggest_correction`: Sugiere correcciones basadas en errores comunes.

### 2.2.6. TokenMap

**Descripción:** Clase que proporciona el mapeo de tokens con números para ser procesados por la tabla de parsing. **Métodos principales:**

- `init_reverse_map`: Inicializa un mapeo inverso para facilitar consultas por código.
- `get_token_code`: Obtiene el código numérico para un tipo de token.
- `get_token_name`: Obtiene el nombre del tipo de token a partir de su código.

### 2.2.7. GLadosDerechos

**Descripción:** Clase que contiene la tabla de lados derechos generada por GikGram y traducida por los estudiantes. Forma parte del módulo Gramática. **Métodos principales:**

- `getLadosDerechos`: Obtiene un símbolo del lado derecho de una regla especificada por número de regla y columna.

### 2.2.8. GNombresTerminales

**Descripción:** Clase que contiene los nombres de los terminales generados por GikGram y traducidos por los estudiantes. **Métodos principales:**

- `getNombresTerminales`: Obtiene el nombre del terminal correspondiente al número especificado.

### 2.2.9. Gramatica

**Descripción:** Clase principal del módulo de gramática que contiene constantes necesarias para el driver de parsing, constantes con rutinas semánticas y métodos para el driver de parsing.

### 2.2.10. Tabla Follows

**Descripción:** Clase encargada de hacer todas las operaciones Follows para el procesamiento de la gramática.

### 2.2.11. Tabla Parsing

**Descripción:** Clase con la tabla de parsing encargada de hacer todo el funcionamiento del mismo. **Constantes principales:**

- **MARCA\_DERECHA:** Código de familia del terminal de fin de archivo.
- **NO\_TERMINAL\_INICIAL:** Número del no-terminal inicial.
- **MAX\_LADO\_DER:** Número máximo de columnas en los lados derechos.
- **MAX\_FOLLOWS:** Número máximo de follows.

**Métodos principales:**

- **esTerminal:** Determina si un símbolo es terminal.
- **esNoTerminal:** Determina si un símbolo es no-terminal.
- **esSimboloSemantico:** Determina si un símbolo es semántico.
- **getTablaParsing:** Obtiene el número de regla desde la tabla de parsing.
- **getLadosDerechos:** Obtiene un símbolo del lado derecho de una regla.
- **getNombresTerminales:** Obtiene el nombre de un terminal.
- **getTablaFollows:** Obtiene el número de terminal del follow del no-terminal.

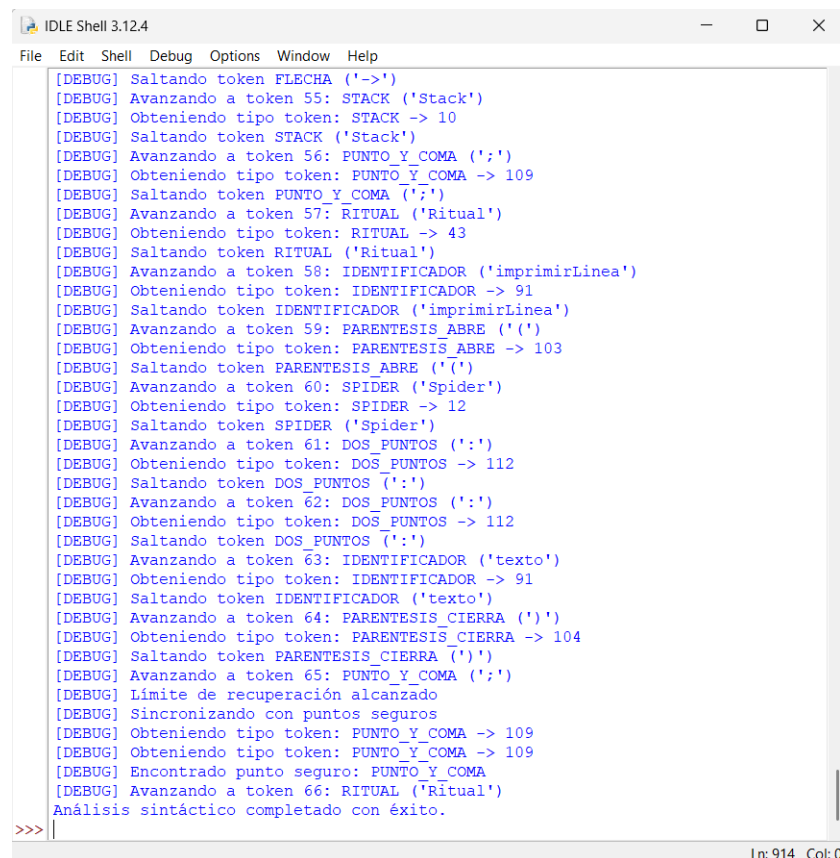
## 2.3. Resultados

Se realizaron pruebas con varios archivos pero documentamos cuatro archivos distintos para validar tanto la funcionalidad general del parser como su capacidad de detección de errores léxicos y sintácticos. Las pruebas se dividen en dos categorías: pruebas válidas (sin errores) y pruebas con errores intencionales.

## Pruebas sin errores

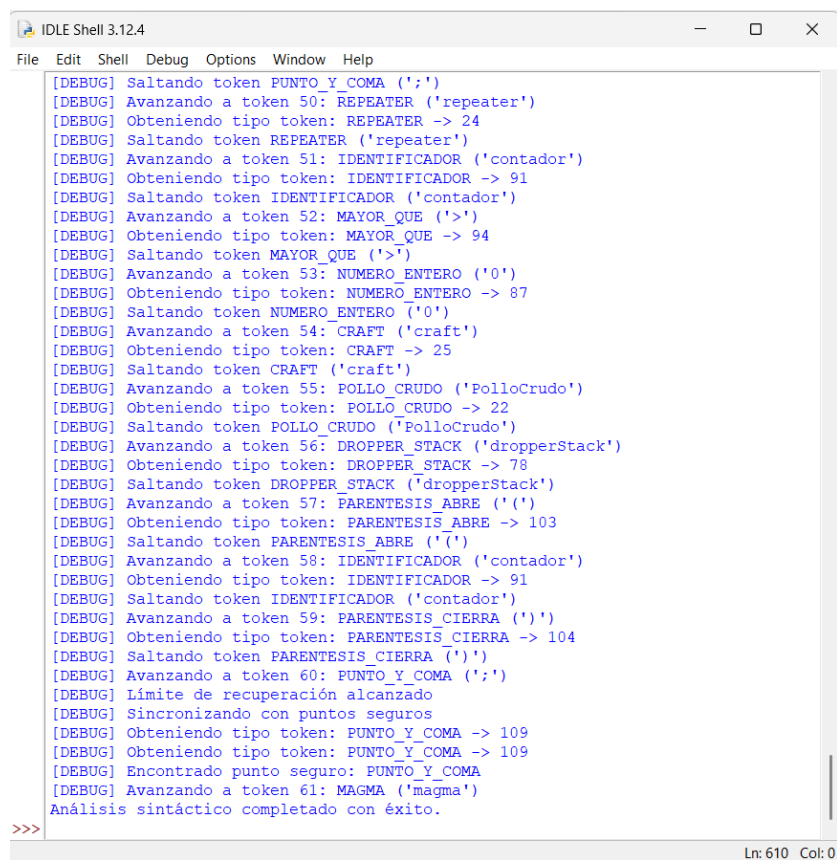
- **07\_Prueba\_PR\_Funciones.txt**: Evalúa el manejo de declaraciones e invocaciones de funciones (**Spell**) y procedimientos (**Ritual**), incluyendo retorno con **respawn**, parámetros múltiples, llamados anidados y estructuras de control dentro del cuerpo de las funciones. Esta prueba pasó sin errores, demostrando que la gramática y el parser reconocen correctamente estructuras complejas de funciones.
- **05\_Prueba\_PR\_Control.txt**: Verifica todas las estructuras de control del lenguaje, incluyendo **repeater**, **target/hit/miss**, **jukebox/disc/silence**, **spawner/exhausted**, **walk/set/to/step** y **with**. El archivo fue aceptado correctamente, confirmando el soporte completo de instrucciones de control en el parser.

## Capturas de pantalla:



```
IDLE Shell 3.12.4
File Edit Shell Debug Options Window Help
[DEBUG] Saltando token FLECHA ('->')
[DEBUG] Avanzando a token 55: STACK ('Stack')
[DEBUG] Obteniendo tipo token: STACK -> 10
[DEBUG] Saltando token STACK ('Stack')
[DEBUG] Avanzando a token 56: PUNTO_Y_COMA (;')
[DEBUG] Obteniendo tipo token: PUNTO_Y_COMA -> 109
[DEBUG] Saltando token PUNTO_Y_COMA (;')
[DEBUG] Avanzando a token 57: RITUAL ('Ritual')
[DEBUG] Obteniendo tipo token: RITUAL -> 43
[DEBUG] Saltando token RITUAL ('Ritual')
[DEBUG] Avanzando a token 58: IDENTIFICADOR ('imprimirLinea')
[DEBUG] Obteniendo tipo token: IDENTIFICADOR -> 91
[DEBUG] Saltando token IDENTIFICADOR ('imprimirLinea')
[DEBUG] Avanzando a token 59: PARENTESIS_ABRE (('')
[DEBUG] Obteniendo tipo token: PARENTESIS_ABRE -> 103
[DEBUG] Saltando token PARENTESIS_ABRE (('')
[DEBUG] Avanzando a token 60: SPIDER ('Spider')
[DEBUG] Obteniendo tipo token: SPIDER -> 12
[DEBUG] Saltando token SPIDER ('Spider')
[DEBUG] Avanzando a token 61: DOS_PUNTOS (':')
[DEBUG] Obteniendo tipo token: DOS_PUNTOS -> 112
[DEBUG] Saltando token DOS_PUNTOS (':')
[DEBUG] Avanzando a token 62: DOS_PUNTOS (':')
[DEBUG] Obteniendo tipo token: DOS_PUNTOS -> 112
[DEBUG] Saltando token DOS_PUNTOS (':')
[DEBUG] Avanzando a token 63: IDENTIFICADOR ('texto')
[DEBUG] Obteniendo tipo token: IDENTIFICADOR -> 91
[DEBUG] Saltando token IDENTIFICADOR ('texto')
[DEBUG] Avanzando a token 64: PARENTESIS_CIERRA ('))
[DEBUG] Obteniendo tipo token: PARENTESIS_CIERRA -> 104
[DEBUG] Saltando token PARENTESIS_CIERRA ('))
[DEBUG] Avanzando a token 65: PUNTO_Y_COMA (;')
[DEBUG] Límite de recuperación alcanzado
[DEBUG] Sincronizando con puntos seguros
[DEBUG] Obteniendo tipo token: PUNTO_Y_COMA -> 109
[DEBUG] Obteniendo tipo token: PUNTO_Y_COMA -> 109
[DEBUG] Encontrado punto seguro: PUNTO_Y_COMA
[DEBUG] Avanzando a token 66: RITUAL ('Ritual')
[DEBUG] Análisis sintáctico completado con éxito.
>>>
```

Figura 2.2: Ejecución exitosa de 07\_Prueba\_PR\_Funciones.txt



```

IDLE Shell 3.12.4
File Edit Shell Debug Options Window Help
[DEBUG] Saltando token PUNTO_Y_COMA (';')
[DEBUG] Avanzando a token 50: REPEATER ('repeater')
[DEBUG] Obteniendo tipo token: REPEATER -> 24
[DEBUG] Saltando token REPEATER ('repeater')
[DEBUG] Avanzando a token 51: IDENTIFICADOR ('contador')
[DEBUG] Obteniendo tipo token: IDENTIFICADOR -> 91
[DEBUG] Saltando token IDENTIFICADOR ('contador')
[DEBUG] Avanzando a token 52: MAYOR_QUE ('>')
[DEBUG] Obteniendo tipo token: MAYOR_QUE -> 94
[DEBUG] Saltando token MAYOR_QUE ('>')
[DEBUG] Avanzando a token 53: NUMERO_ENTERO ('0')
[DEBUG] Obteniendo tipo token: NUMERO_ENTERO -> 87
[DEBUG] Saltando token NUMERO_ENTERO ('0')
[DEBUG] Avanzando a token 54: CRAFT ('craft')
[DEBUG] Obteniendo tipo token: CRAFT -> 25
[DEBUG] Saltando token CRAFT ('craft')
[DEBUG] Avanzando a token 55: POLLO_CRUDO ('PolloCrudo')
[DEBUG] Obteniendo tipo token: POLLO_CRUDO -> 22
[DEBUG] Saltando token POLLO_CRUDO ('PolloCrudo')
[DEBUG] Avanzando a token 56: DROPPER_STACK ('dropperStack')
[DEBUG] Obteniendo tipo token: DROPPER_STACK -> 78
[DEBUG] Saltando token DROPPER_STACK ('dropperStack')
[DEBUG] Avanzando a token 57: PARENTESIS_ABRE '('')
[DEBUG] Obteniendo tipo token: PARENTESIS_ABRE -> 103
[DEBUG] Saltando token PARENTESIS_ABRE '('')
[DEBUG] Avanzando a token 58: IDENTIFICADOR ('contador')
[DEBUG] Obteniendo tipo token: IDENTIFICADOR -> 91
[DEBUG] Saltando token IDENTIFICADOR ('contador')
[DEBUG] Avanzando a token 59: PARENTESIS_CIERRA ('')')
[DEBUG] Obteniendo tipo token: PARENTESIS_CIERRA -> 104
[DEBUG] Saltando token PARENTESIS_CIERRA ('')')
[DEBUG] Avanzando a token 60: PUNTO_Y_COMA (';')
[DEBUG] Límite de recuperación alcanzado
[DEBUG] Sincronizando con puntos seguros
[DEBUG] Obteniendo tipo token: PUNTO_Y_COMA -> 109
[DEBUG] Obteniendo tipo token: PUNTO_Y_COMA -> 109
[DEBUG] Encontrado punto seguro: PUNTO_Y_COMA
[DEBUG] Avanzando a token 61: MAGMA ('magma')
Análisis sintáctico completado con éxito.
>>>
Ln: 610 Col: 0

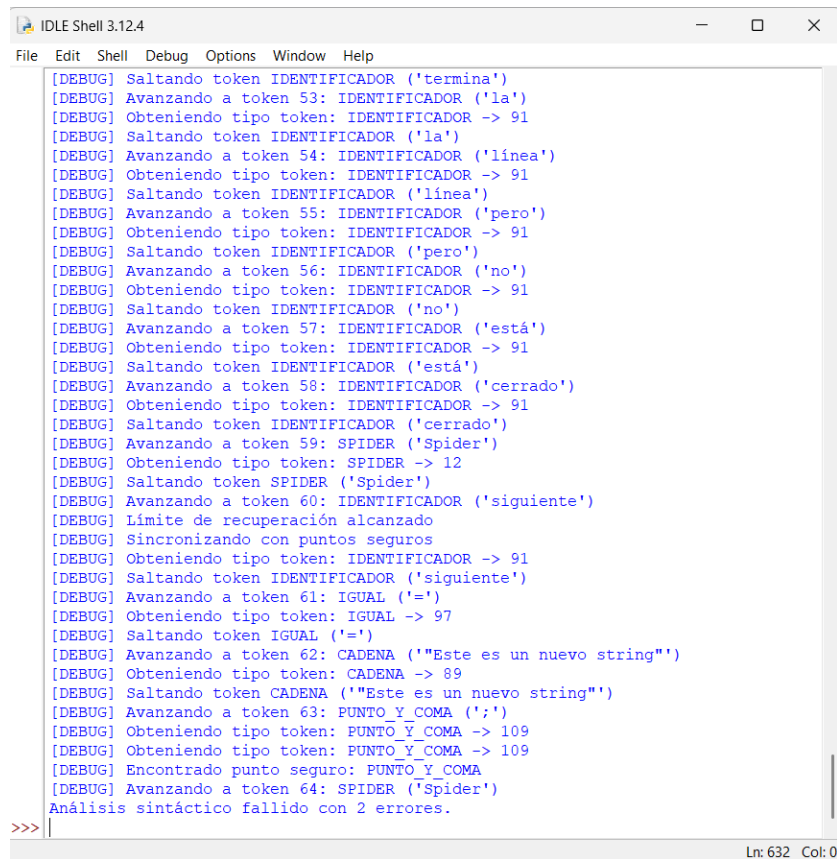
```

Figura 2.3: Ejecución exitosa de 05\_Prueba\_PR\_Control.txt

## Pruebas con errores detectados

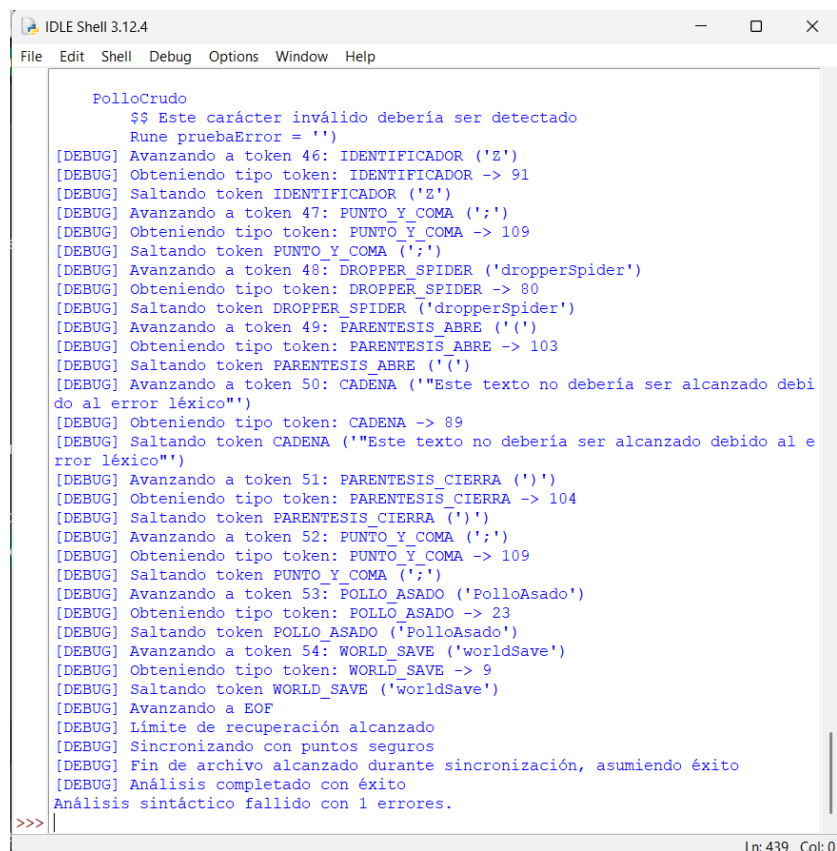
- **35\_Prueba\_Err\_StringNoTerminado.txt**: Contiene múltiples casos de cadenas de texto mal formadas (sin comilla de cierre, seguidas por comentarios o nuevos tokens). El analizador léxico identificó correctamente los errores de forma y generó mensajes adecuados, además de mostrar recuperación parcial al continuar con la ejecución del archivo.
- **37\_Prueba\_Err\_CaracterNoTerminado.txt**: Se incluyen literales de carácter mal contruidos (sin cierre, vacíos o con múltiples símbolos). Todos los casos fueron detectados por el analizador léxico como errores léxicos válidos. Además, el parser evitó errores en cascada, gracias a la estrategia de recuperación implementada.

Capturas de pantalla:



```
File Edit Shell Debug Options Window Help
[DEBUG] Saltando token IDENTIFICADOR ('termina')
[DEBUG] Avanzando a token 53: IDENTIFICADOR ('la')
[DEBUG] Obteniendo tipo token: IDENTIFICADOR -> 91
[DEBUG] Saltando token IDENTIFICADOR ('la')
[DEBUG] Avanzando a token 54: IDENTIFICADOR ('línea')
[DEBUG] Obteniendo tipo token: IDENTIFICADOR -> 91
[DEBUG] Saltando token IDENTIFICADOR ('línea')
[DEBUG] Avanzando a token 55: IDENTIFICADOR ('pero')
[DEBUG] Obteniendo tipo token: IDENTIFICADOR -> 91
[DEBUG] Saltando token IDENTIFICADOR ('pero')
[DEBUG] Avanzando a token 56: IDENTIFICADOR ('no')
[DEBUG] Obteniendo tipo token: IDENTIFICADOR -> 91
[DEBUG] Saltando token IDENTIFICADOR ('no')
[DEBUG] Avanzando a token 57: IDENTIFICADOR ('está')
[DEBUG] Obteniendo tipo token: IDENTIFICADOR -> 91
[DEBUG] Saltando token IDENTIFICADOR ('está')
[DEBUG] Avanzando a token 58: IDENTIFICADOR ('cerrado')
[DEBUG] Obteniendo tipo token: IDENTIFICADOR -> 91
[DEBUG] Saltando token IDENTIFICADOR ('cerrado')
[DEBUG] Avanzando a token 59: SPIDER ('Spider')
[DEBUG] Obteniendo tipo token: SPIDER -> 12
[DEBUG] Saltando token SPIDER ('Spider')
[DEBUG] Avanzando a token 60: IDENTIFICADOR ('siguiente')
[DEBUG] Límite de recuperación alcanzado
[DEBUG] Sincronizando con puntos seguros
[DEBUG] Obteniendo tipo token: IDENTIFICADOR -> 91
[DEBUG] Saltando token IDENTIFICADOR ('siguiente')
[DEBUG] Avanzando a token 61: IGUAL ('=')
[DEBUG] Obteniendo tipo token: IGUAL -> 97
[DEBUG] Saltando token IGUAL ('=')
[DEBUG] Avanzando a token 62: CADENA ("Este es un nuevo string")
[DEBUG] Obteniendo tipo token: CADENA -> 89
[DEBUG] Saltando token CADENA ("Este es un nuevo string")
[DEBUG] Avanzando a token 63: PUNTO_Y_COMA (;)
[DEBUG] Obteniendo tipo token: PUNTO_Y_COMA -> 109
[DEBUG] Obteniendo tipo token: PUNTO_Y_COMA -> 109
[DEBUG] Encontrado punto seguro: PUNTO_Y_COMA
[DEBUG] Avanzando a token 64: SPIDER ('Spider')
>>> Análisis sintáctico fallido con 2 errores.
```

Figura 2.4: Errores léxicos detectados en 35\_Prueba\_Err\_StringNoTerminado.txt



```
PolloCrudo
$$ Este carácter inválido debería ser detectado
Rune pruebaError = '')
[DEBUG] Avanzando a token 46: IDENTIFICADOR ('Z')
[DEBUG] Obteniendo tipo token: IDENTIFICADOR -> 91
[DEBUG] Saltando token IDENTIFICADOR ('Z')
[DEBUG] Avanzando a token 47: PUNTO_Y_COMA (;')
[DEBUG] Obteniendo tipo token: PUNTO_Y_COMA -> 109
[DEBUG] Saltando token PUNTO_Y_COMA (;')
[DEBUG] Avanzando a token 48: DROPPER_SPIDER ('dropperSpider')
[DEBUG] Obteniendo tipo token: DROPPER_SPIDER -> 80
[DEBUG] Saltando token DROPPER_SPIDER ('dropperSpider')
[DEBUG] Avanzando a token 49: PARENTESIS_ABRE (('')
[DEBUG] Obteniendo tipo token: PARENTESIS_ABRE -> 103
[DEBUG] Saltando token PARENTESIS_ABRE (('')
[DEBUG] Avanzando a token 50: CADENA ("Este texto no debería ser alcanzado debi
do al error léxico")
[DEBUG] Obteniendo tipo token: CADENA -> 89
[DEBUG] Saltando token CADENA ("Este texto no debería ser alcanzado debido al e
rror léxico")
[DEBUG] Avanzando a token 51: PARENTESIS_CIERRA (')')
[DEBUG] Obteniendo tipo token: PARENTESIS_CIERRA -> 104
[DEBUG] Saltando token PARENTESIS_CIERRA (')')
[DEBUG] Avanzando a token 52: PUNTO_Y_COMA (;')
[DEBUG] Obteniendo tipo token: PUNTO_Y_COMA -> 109
[DEBUG] Saltando token PUNTO_Y_COMA (;')
[DEBUG] Avanzando a token 53: POLLO_ASADO ('PolloAsado')
[DEBUG] Obteniendo tipo token: POLLO_ASADO -> 23
[DEBUG] Saltando token POLLO_ASADO ('PolloAsado')
[DEBUG] Avanzando a token 54: WORLD_SAVE ('worldSave')
[DEBUG] Obteniendo tipo token: WORLD_SAVE -> 9
[DEBUG] Saltando token WORLD_SAVE ('worldSave')
[DEBUG] Avanzando a EOF
[DEBUG] Límite de recuperación alcanzado
[DEBUG] Sincronizando con puntos seguros
[DEBUG] Fin de archivo alcanzado durante sincronización, asumiendo éxito
[DEBUG] Análisis completado con éxito
Análisis sintáctico fallido con 1 errores.
>>>
```

Figura 2.5: Errores léxicos detectados en 37\_Prueba\_Err\_CaracterNoTerminado.txt

En resumen, las pruebas demuestran que el parser reconoce correctamente estructuras válidas complejas y que también maneja adecuadamente errores léxicos, incluyendo múltiples casos problemáticos seguidos, sin caer en errores en cascada. Esto valida tanto la gramática como el driver de parsing implementado.