



Instituto Tecnológico de Costa Rica

Escuela de Computación

Compiladores e Intérpretes, IC5701

Etapas Cuatro: Generador de Código

Estudiantes:

Samir Cabrera Tabash, 2022161229

Luis Urbina Salazar, 2023156802

Primer semestre del año 2025

Índice general

Índice general	1
1. Definición del Lenguaje	8
1.1. Logo y nombre del Lenguaje	8
1.2. Elementos Fundamentales del lenguaje	9
1.3. Sistemas de Asignación	11
1.4. Tipos de Datos	12
1.5. Literales	15
1.6. Sistemas de Acceso a Datos	18
1.7. Operadores y Expresiones	19
1.8. Estructuras de Control	23
1.9. Funciones y Procedimientos	29
1.10. Elementos Auxiliares	33
1.11. Listado de Palabras Reservadas	38
2. Gramatica del Lenguaje	40
2.1. Estructura Básica del Programa - Gramática BNF	40
2.2. Sistemas de Asignación - Gramática BNF	45
2.3. Tipos de Datos - Gramática BNF	49
2.4. Literales - Gramática BNF	56
2.5. Sistemas de Acceso - Gramática BNF	65
2.6. Operadores - Gramática BNF	71
2.7. Estructuras de Control - Gramática BNF	77
2.8. Funciones y Procedimientos - Gramática BNF	84
2.9. Elementos Auxiliares - Gramática BNF	89
3. Algoritmos de Conversión	95
3.1. Introducción	95
3.2. Tipos de Datos Básicos	95
3.3. Matriz de Conversiones	96
3.4. Conversiones desde Stack	96

3.5.	Conversiones desde Ghist	99
3.6.	Conversiones desde Torch	101
3.7.	Conversiones desde Rune	103
3.8.	Conversiones desde Spider	105
3.9.	Conversiones desde Creativo	108
4.	Listado de Errores Léxicos	111
4.1.	Introducción	111
4.2.	Tabla de Errores Léxicos	111
4.3.	Recuperación de Errores Críticos	113
4.4.	Recuperación de Error para Evitar Errores en Cascada	114
5.	Documentacion del Automata	116
5.1.	Introducción	116
5.2.	Consideraciones Técnicas	116
5.3.	Diagrama Principal (Main)	116
5.3.1.	Descripción	116
5.3.2.	Características	117
5.3.3.	Lógica de Enrutamiento	117
5.4.	Reconocedor de Números	119
5.4.1.	Descripción	119
5.4.2.	Tipos Numéricos	120
5.5.	Reconocedor de Strings y Caracteres	121
5.5.1.	Descripción	121
5.5.2.	Características	121
5.6.	Reconocedor de Operadores y Símbolos	122
5.6.1.	Descripción	122
5.6.2.	Categorías de Operadores	122
5.7.	Reconocedor de Comentarios	124
5.7.1.	Descripción	124
5.7.2.	Estados	124
5.8.	Reconocedor de Literales Especiales	125
5.8.1.	Descripción	125
5.8.2.	Tipos de Literales	125
5.9.	Reconocedor de Identificadores	128
5.9.1.	Descripción	128
5.9.2.	Reglas de Identificadores	128
5.10.	Reconocedor de Palabras Reservadas	129
5.10.1.	Propósito	129
5.10.2.	Características Principales	130
5.10.3.	Palabras Implementadas	130

5.10.4. Comportamiento	130
5.10.5. Restricciones	130
5.10.6. Diagrama	131
6. Documentacion del Scanner	145
6.1. Conceptos básicos del Scanner	145
6.1.1. Estructura General	145
6.1.2. Funcionamiento del Scanner	146
6.1.3. Recuperación de Errores	147
6.1.4. Reconocimiento de Comentarios	147
6.1.5. Generación de Resultados	147
6.1.6. Cumplimiento de las Reglas del Proyecto	148
6.2. Core del Scanner	148
6.2.1. Clase Scanner	148
6.2.2. Clase ErrorHandler	149
6.2.3. Métodos Principales del Scanner	149
6.2.4. Métodos Auxiliares	149
6.2.5. Funcionamiento del Análisis Léxico	150
6.2.6. Clase IntegratedAutomaton	150
6.2.7. Recuperación de Errores	151
6.2.8. Importancia del Scanner	151
6.3. Explicación del Autómata	151
6.3.1. Estructura Base del Autómata Integrado	151
6.3.2. Procesamiento por Tipo de Token	152
6.3.3. Procesamiento de Comentarios	152
6.3.4. Procesamiento de Strings y Caracteres	153
6.3.5. Procesamiento de Números	153
6.3.6. Procesamiento de Identificadores	153
6.3.7. Procesamiento de Operadores y Símbolos	154
6.3.8. Procesamiento de Delimitadores de Bloque	154
6.3.9. Flujo de Procesamiento	154
6.3.10. Manejo de Errores	155
6.4. Explicación Tokenizador	155
6.4.1. Clase Token Mejorada	155
6.4.2. Categorías de Tokens	156
6.4.3. Palabras Reservadas	158
6.4.4. Manejo de Operadores	158
6.5. Explicación de Brickwall	159
6.5.1. Descripción general	159
6.5.2. Parámetros de entrada	159
6.5.3. Estructura de salida	160

6.5.4.	Elementos visuales	160
6.5.5.	Estadísticas generadas	161
6.5.6.	Implementación	161
6.5.7.	Uso típico	161
6.6.	Ejecución del Scanner	162
6.6.1.	Explicación de Scripts	162
6.6.2.	Ejecución del Scanner	163
7.	Documentacion del Parser	168
7.1.	Documentacion inicial	168
7.2.	Gramática del Parser	168
7.2.1.	Clase Parser	187
7.2.2.	Funciones auxiliares	189
7.2.3.	Características relevantes	189
7.2.4.	SpecialTokens	189
7.2.5.	TokenMap	190
7.2.6.	GLadosDerechos	190
7.2.7.	GNombresTerminales	190
7.2.8.	Gramatica	191
7.2.9.	Tabla Follows	191
7.2.10.	Tabla Parsing	191
7.3.	Resultados	192
8.	Documentación del Análisis Semántico	197
8.1.	Diccionario Semantico	199
8.1.1.	Valor Tipo	199
8.1.2.	Chequeo de existencia de identificador	200
8.1.3.	Chequeo de constantes	200
8.1.4.	Chequeo de Overflow	201
8.1.5.	Chequeo de Pollo	202
8.1.6.	Chequeo de Shelf Size	202
8.1.7.	Chequeo de tipo de operación	203
8.1.8.	Chequeo de Uso de Variables	203
8.1.9.	Chequeo de WorldName	204
8.1.10.	Chequeo de WorldSave	204
8.1.11.	Chequeo de Nombre de Archivo	205
8.1.12.	Chequeo de Igualdad de Operadores	205
8.1.13.	Chequeo de Inicialización de Variables	206
8.1.14.	Chequeo de División por Cero	206
8.1.15.	Chequeo de Estructura Target y Hit Miss	207
8.2.	Manejo de Tipo de Datos	208

8.2.1.	Manejo de Tipos STACK , GHOST , TORCH , SPIDER y RUNE	208
8.2.2.	Manejo de Tipo BOOK	209
8.2.3.	Manejo de Tipo CHEST	209
8.2.4.	Manejo de Tipo SHELF	210
8.2.5.	Manejo de Tipo ENTITY	210
8.2.6.	Manejo de Tipo SPELL	211
8.2.7.	Manejo de Tipo RITUAL	211
8.3.	Explicación del Código Semántico	213
8.3.1.	Modificación del Parser	213
8.3.2.	Símbolos	213
8.3.3.	Verificación Pasada de Tokens	214
8.3.4.	Verificación Futura de Tokens	214
8.3.5.	Historial Semántico	215
8.3.6.	Errores Terminales	215
8.4.	Explicación de la Tabla Semántica	217
8.4.1.	Estructura de la Tabla Semántica	217
8.4.2.	Análisis de Funcionamiento	217
8.4.3.	Resultados de la Tabla de Símbolos	218
8.4.4.	Ejemplos reales de la tabla semantica	219
8.5.	Resultados de Semantica	220
8.5.1.	Resultados de la tabla de hash	221
8.5.2.	Resultados de historial semantico	222
8.5.3.	Resultados de historial semantico negativo	224
9.	Generación de Código	225
9.1.	Introducción	225
9.1.1.	Objetivos de la Generación de Código	225
9.1.2.	Arquitectura del Generador	226
9.1.3.	Estrategia de Traducción	227
9.1.4.	Manejo de Errores en Generación	227
9.1.5.	Integración con Etapas Anteriores	228
9.2.	Análisis de Código	228
9.2.1.	Chequeo de Variables No Inicializadas	228
9.2.2.	Chequeo de Código Muerto	229
9.2.3.	Chequeo de Tipos en Operaciones Binarias	229
9.2.4.	Chequeo de Límites en SHELF	230
9.2.5.	Chequeo de Constantes Reasignadas	230
9.2.6.	Chequeo de Argumentos en Llamadas a Funciones	231
9.2.7.	Chequeo de Retorno en Funciones	231
9.2.8.	Chequeo de Archivos No Cerrados	232
9.2.9.	Chequeo de Bucles Infinitos	232

9.2.10.	Chequeo de Shadowing de Variables	233
9.2.11.	Chequeo de Conversiones Implícitas Peligrosas	233
9.2.12.	Chequeo de Recursos sin Liberar	234
9.2.13.	Chequeo de Pre/Post Condiciones	234
9.2.14.	Chequeo de Consistencia en WORLDNAME/WORLD- SAVE	235
9.2.15.	Chequeo de Rendimiento	235
9.3.	Runtime Library	236
9.3.1.	Introducción a la Runtime Library	236
9.3.2.	Arquitectura de la Runtime Library	236
9.3.3.	Implementación de Operaciones	237
9.3.4.	Operaciones de Cadenas	239
9.3.5.	Generación Automática	239
9.3.6.	Manejo de Errores en Runtime	239
9.3.7.	Optimizaciones Implementadas	240
9.3.8.	Compatibilidad y Portabilidad	241
9.4.	Ejecución de Programa	241
9.4.1.	Introducción al Proceso de Ejecución	241
9.4.2.	Arquitectura de Ejecución	242
9.4.3.	Modelo de Ejecución	242
9.4.4.	Integración con Runtime Library	243
9.4.5.	Manejo de Entrada/Salida	244
9.4.6.	Control de Flujo en Ejecución	244
9.4.7.	Terminación del Programa	245
9.4.8.	Optimizaciones de Ejecución	246
9.4.9.	Depuración y Diagnóstico	246
9.4.10.	Compatibilidad y Portabilidad	247
9.5.	Prueba de Implementacion	248
9.5.1.	Pregunta 1	248
9.5.2.	Pregunta 2	251
9.6.	Implementación del Generador de Código	253
9.6.1.	Introducción a la Implementación	253
9.6.2.	Arquitectura de Clases	253
9.6.3.	Proceso de Generación de Runtime Library	254
9.6.4.	Generación de Código Principal	255
9.6.5.	Archivos de Prueba y Validación	257
9.6.6.	Control de Flujo Avanzado	259
9.6.7.	Integración y Compatibilidad	259
9.6.8.	Manejo de Errores y Robustez	260
9.6.9.	Optimizaciones y Eficiencia	261
9.6.10.	Extensibilidad y Mantenimiento	261

9.6.11. Casos de Uso y Ejemplos Prácticos	262
9.6.12. Integración con el Sistema Completo	262

Capítulo 1

Definición del Lenguaje

1.1. Logo y nombre del Lenguaje

NOTCH ENGINE



Justificación: Notch Engine es un nombre que rinde homenaje a Markus "Notch" Persson, el creador original de Minecraft. El término "*Engine*" refleja que este lenguaje funciona como un motor que impulsa la creación y ejecución de programas basados en la temática de Minecraft, permitiendo a los usuarios construir soluciones de programación con la misma creatividad y libertad que ofrece el juego. Al igual que Minecraft evolucionó de un simple concepto a un universo complejo, Notch Engine proporciona las herramientas fundamentales para que los programadores construyan desde simples scripts hasta aplicaciones complejas utilizando elementos familiares del mundo de Minecraft.

Extensión de archivo: .ne

1.2. Elementos Fundamentales del lenguaje

Estructura del título del programa

WorldName <id>:

Representa la definición del nombre del nuevo mundo que se crea. En Minecraft, cada mundo tiene un nombre único que lo identifica y lo diferencia de otros mundos creados. De manera similar, en Notch Engine cada programa debe definir un nombre de mundo que servirá como identificador del proyecto.

Ejemplo:

WorldName MiProyecto:

\$\$ Código del programa

Sección de Constantes (Bedrock)

Bedrock

Representa la base indestructible y fundamental del programa. Las constantes, como el bedrock en Minecraft, son valores inmutables que una vez definidos no pueden ser alterados durante la ejecución. En el juego, el bedrock es el único bloque que no puede ser destruido en modo supervivencia, simbolizando así la inmutabilidad de las constantes en nuestro lenguaje.

Ejemplo:

Bedrock

\$\$ Declaraciones de constantes

Sección de Tipos (ResourcePack)

ResourcePack: <id>: <tipo-id>

Similar a cómo los resource packs en Minecraft modifican la apariencia y funcionalidad del juego, esta sección define los diferentes tipos de datos y sus conversiones que podrán ser utilizados en el programa. Los resource packs permiten personalizar la experiencia del juego, así como los tipos personalizados permiten adaptar el lenguaje a las necesidades del programador.

Ejemplo:

ResourcePack:

\$\$ Declaración de tipos

Sección de Variables (Inventory)

Inventory

En Minecraft, el inventario es el espacio donde el jugador almacena y organiza los objetos que recolecta. De manera análoga, esta sección define el espacio donde se declaran las variables que almacenarán los valores durante la ejecución del programa, permitiendo su organización y acceso.

Ejemplo:

Inventory

\$\$ Declaraciones de variables

Sección de Prototipos (Recipe)

Recipe

Las recetas en Minecraft son instrucciones que permiten craftear objetos o bloques a partir de componentes más básicos. En programación, los prototipos definen la estructura de las funciones y procedimientos antes de su implementación completa, estableciendo qué ingredientes (parámetros) se necesitan para crear un resultado específico.

Ejemplo:

Recipe

\$\$ Declaraciones de prototipos

Sección de Rutinas (CraftingTable)

CraftingTable

La mesa de craftero en Minecraft permite crear nuevos objetos siguiendo recetas específicas. En nuestro lenguaje, esta sección es donde se implementan las funciones y procedimientos definidos previamente como prototipos, creando nuevas funcionalidades a partir de código más básico.

Ejemplo:

CraftingTable

\$\$ Implementación de rutinas

Punto de Entrada (SpawnPoint)

SpawnPoint

El punto de spawn es donde comienza el jugador al entrar a un mundo de Minecraft. De manera similar, esta sección define el punto inicial de ejecución del programa, donde comienza el flujo de instrucciones.

Ejemplo:

```
SpawnPoint
  $$ Código principal
```

1.3. Sistemas de Asignación

Sistema de Asignación de Constantes (Obsidian)

Obsidian <tipo><id><value>

El obsidian (obsidiana) es uno de los materiales más duros en Minecraft, resistente incluso a las explosiones. Esta característica lo convierte en el símbolo perfecto para representar las constantes, valores que una vez definidos no pueden ser modificados durante la ejecución del programa. La declaración de constantes sigue una estructura clara donde se especifica el tipo, identificador y valor.

Ejemplo:

```
Bedrock
  Obsidian Stack MAX_LEVEL 100;
```

Sistema de Asignación de Tipos (Anvil)

Anvil <id>-><tipo>

El yunque en Minecraft sirve para reparar y modificar objetos, combinando diferentes items para mejorarlos o transformarlos. De manera similar, el sistema de asignación de tipos permite definir conversiones entre diferentes tipos de datos, estableciendo cómo se transformará un tipo en otro durante la ejecución del programa.

Ejemplo:

```
ResourcePack:
  Anvil Ghast -> Stack;
```

Sistema de Declaración de Variables

```
<tipo>id = <lit>, id = <lit>
```

La inicialización y declaración múltiple son características que el lenguaje provee, aunque son opcionales. El sistema utiliza el símbolo "-" para la asignación, siendo consistente con muchos lenguajes de programación modernos y facilitando la comprensión del código.

Ejemplo:

Inventory

```
Stack level = 5;  
Spider name = "Steve";  
Torch isActive = On, hasItems = Off;
```

1.4. Tipos de Datos

Tipo de Dato Entero (Stack)

Stack

Los enteros en Notch Engine se representan mediante el tipo **Stack**. Esta denominación hace referencia a cómo los objetos en Minecraft pueden apilarse en bloques, formando una estructura ordenada con valores discretos y contables. Al igual que las pilas de bloques en el juego, los enteros representan cantidades exactas y completas.

Ejemplo:

Inventory

```
Stack playerLevel = 30;  
Stack blockCount = -5;
```

Tipo de Dato Caracter (Rune)

Rune

Los caracteres individuales se representan mediante el tipo **Rune**. En Minecraft, las runas son símbolos usados en encantamientos, cada uno con un significado específico. De manera similar, este tipo de dato almacena un único símbolo del lenguaje, como letras, números o símbolos especiales.

Ejemplo:

Inventory

```
Rune initial = 'K';
```

Tipo de Dato String (Spider)

Spider

Las cadenas de caracteres se representan mediante el tipo **Spider**. El nombre hace referencia a que las arañas en Minecraft cuando son eliminadas dejan una string (hilo). Este tipo de dato permite almacenar secuencias de caracteres para representar texto.

Ejemplo:

Inventory

```
Spider playerName = "Steve";
```

Tipo de Dato Booleano (Torch)

Torch

Los valores booleanos se representan mediante el tipo **Torch**. Al igual que una antorcha en Minecraft puede estar encendida o apagada, este tipo de dato puede tener uno de dos estados: **On** (verdadero) o **Off** (falso).

Ejemplo:

Inventory

```
Torch isAlive = On;  
Torch hasItems = Off;
```

Tipo de Dato Conjunto (Chest)

Chest

Los conjuntos se representan mediante el tipo **Chest**. Un cofre en Minecraft puede almacenar múltiples objetos únicos, de manera similar, este tipo de dato permite almacenar colecciones de elementos sin repeticiones.

Ejemplo:

Inventory

```
Chest vowels = { : 'a', 'e', 'i', 'o', 'u' :};
```

Tipo de Dato Archivo de Texto (Book)

Book

Los archivos de texto se representan mediante el tipo **Book**. Como los libros en Minecraft, este tipo de dato permite almacenar y acceder a información textual persistente entre ejecuciones del programa.

Ejemplo:

Inventory

```
Book logFile = {/ "gameLog.txt", 'E' /};
```

Tipo de Dato Números Flotantes (Ghast)

Ghast

Los números de punto flotante se representan mediante el tipo **Ghast**. Esta denominación hace referencia a los Ghasts de Minecraft, criaturas que flotan en el aire, al igual que los números flotantes representan valores que incluyen una parte decimal.

Ejemplo:

Inventory

```
Ghast pi = 3.14;  
Ghast temperature = -3.5;
```

Tipo de Dato Arreglos (Shelf)

Shelf

Los arreglos se representan mediante el tipo **Shelf**. Una estantería en Minecraft organiza libros de manera ordenada, donde cada libro ocupa un lugar específico. De manera similar, los arreglos en Notch Engine permiten almacenar elementos del mismo tipo en posiciones indexadas.

Ejemplo:

Inventory

```
Shelf Stack scores = [1, 2, 3, 4, 5];
```

Tipo de Dato Registros (Entity)

Entity

Los registros se representan mediante el tipo **Entity**. Una entidad en Minecraft encapsula múltiples atributos y comportamientos relacionados. De manera similar, este tipo de dato permite agrupar diversos campos de diferentes tipos bajo una misma estructura.

Ejemplo:

```
Entity Player
  Spider name;
  Stack level;
  Ghast health;
```

Inventory

```
Entity Player steve = {name: "Steve", level: 30, health: 20.0};
```

1.5. Literales

Literales Booleanas (On/Off)

On/Off

Los valores booleanos literales en Notch Engine se representan mediante las palabras clave **On** y **Off**, que corresponden a verdadero y falso respectivamente. Esta notación es intuitiva y mantiene la coherencia con la metáfora de una antorcha (**Torch**) que puede estar encendida o apagada.

Ejemplo:

Inventory

```
Torch isActive = On;
Torch isDone = Off;
```

Literales de Conjuntos

{: elemento1, elemento2, ... :}

Los conjuntos literales se delimitan mediante los símbolos **{: y :}**, que encierran los elementos del conjunto separados por comas. Esta notación utiliza los corchetes convencionales pero les agrega los dos puntos para diferenciarlos claramente de otros tipos de agrupaciones como los registros.

Ejemplo:

Inventory

```
Chest vowels = { : 'a', 'e', 'i', 'o', 'u' :};  
Chest tools = { : "pickaxe", "shovel", "axe" :};
```

Literales de Archivos

{/ "nombre_archivo", 'modo' /}

Los literales de archivos se delimitan mediante los símbolos {/ y /}, que encierran el nombre del archivo como cadena y el modo de acceso como un carácter. Los modos disponibles son 'L' para lectura, 'E' para escritura y 'A' para actualización.

Ejemplo:

Inventory

```
Book logFile = {/ "gameLog.txt", 'E' /};  
Book configFile = {/ "config.dat", 'L' /};
```

Literales de Números Flotantes

-3.14

Los números flotantes literales siguen la notación decimal convencional, con un punto separando la parte entera de la decimal. Pueden incluir signo negativo si es necesario.

Ejemplo:

Inventory

```
Ghast pi = 3.14159;  
Ghast gravity = -9.81;
```

Literales de Enteros

5, -5

Los enteros literales se representan con dígitos decimales y pueden incluir signo negativo si es necesario.

Ejemplo:

Inventory

```
Stack level = 50;  
Stack depth = -64;
```

Literales de Caracteres

'K'

Los caracteres literales se encierran entre comillas simples y pueden representar cualquier símbolo individual, como letras, dígitos o caracteres especiales.

Ejemplo:

Inventory

```
Rune grade = 'A';  
Rune symbol = '*';
```

Literales de Strings

"string"

Las cadenas de texto literales se encierran entre comillas dobles y pueden contener cualquier secuencia de caracteres.

Ejemplo:

Inventory

```
Spider greeting = "Hello, miner!";  
Spider message = "Welcome to Notch Engine";
```

Literales de Arreglos

[1, 2, 3, 4, 5]

Los arreglos literales se delimitan mediante corchetes y contienen elementos del mismo tipo separados por comas.

Ejemplo:

Inventory

```
Shelf Stack scores = [1, 2, 3, 4, 5];  
Shelf Spider names = ["Steve", "Alex", "Herobrine"];
```

Literales de Registros

{<id>: <value>, <id>: <value>, ...}

Los registros literales se delimitan mediante llaves y contienen pares campo-valor separados por comas, donde cada par consiste en un identificador seguido de dos puntos y un valor.

Ejemplo:

Inventory

```
Entity Player steve = {name: "Steve", level: 30, health: 20.0};  
Entity Block dirt = {type: "Dirt", hardness: 0.5, transparent: Off};
```

1.6. Sistemas de Acceso a Datos

Sistema de Acceso Arreglos

[2][3][4]

El acceso a elementos de arreglos en Notch Engine utiliza la notación de corchetes para cada dimensión, similar a lenguajes como C y Java. A diferencia de otras implementaciones, Notch Engine solo admite esta forma de acceso y no permite la notación condensada con comas como [2,3,4].

Ejemplo:

Inventory

```
Shelf Stack matrix = [[1, 2], [3, 4]];  
Stack value = matrix[0][1]; $$ Accede al valor 2
```

Sistema de Acceso Strings

string[1]

El acceso a caracteres individuales dentro de una cadena de texto se realiza mediante la notación de corchetes, donde el índice indica la posición del carácter deseado (comenzando desde 0). Esta sintaxis es similar a cómo se accede a caracteres individuales en lenguajes como C.

Ejemplo:

Inventory

```
Spider name = "Steve";  
Rune firstChar = name[0]; $$ Obtiene 'S'
```

Sistema de Acceso Registros

@ (e.g.: registro@campo)

El acceso a campos individuales dentro de un registro se realiza mediante el operador '@', seguido del nombre del campo. Esta notación es diferente a la usada en otros lenguajes (que suelen usar el punto) y proporciona una sintaxis visual clara para distinguir el acceso a campos de registros.

Ejemplo:

```
Entity Player
  Spider name;
  Stack level;
  Torch hasItems;
```

```
Inventory
  Entity Player steve;
  steve@name = "Steve";
  Stack playerLevel = steve@level;
```

Asignación y Familia

= (i.e: =, +=, -=, *=, /=, %=)

Notch Engine provee una familia completa de operadores de asignación, similares a los encontrados en lenguajes como C. Estos incluyen la asignación simple (=), así como operadores compuestos que combinan una operación aritmética con la asignación (+=, -=, *=, /=, %=).

Ejemplo:

```
Inventory
  Stack counter = 10;
  counter += 5;      $$ counter = counter + 5
  counter -= 2;      $$ counter = counter - 2
  counter *= 3;      $$ counter = counter * 3
  counter /= 2;      $$ counter = counter / 2
  counter %= 4;      $$ counter = counter % 4
```

1.7. Operadores y Expresiones

Operaciones aritméticas básicas de enteros

+, -, *, //, %

Notch Engine proporciona los operadores aritméticos básicos para manipular valores enteros. Estos incluyen suma (+), resta (-), multiplicación (*), división entera (//) y módulo (%). Es importante notar que la división de enteros utiliza dos barras diagonales (//) en lugar de una, lo que la diferencia de otros lenguajes como C.

Ejemplo:

Inventory

```
Stack a = 10 + 5;    $$ Suma: 15
Stack b = 10 - 5;    $$ Resta: 5
Stack c = 10 * 5;    $$ Multiplicación: 50
Stack d = 10 // 3;    $$ División entera: 3
Stack e = 10 % 3;    $$ Módulo (resto): 1
```

Incremento y Decremento

soulsand, magma

Las operaciones de incremento y decremento se representan mediante **soulsand** y **magma** respectivamente. Estos nombres hacen referencia a cómo, en Minecraft, al combinarse con agua, el bloque **soulsand** hace subir al jugador mientras que el bloque **magma** lo hace bajar. Estas operaciones permiten aumentar o disminuir en una unidad el valor de una variable.

Ejemplo:

Inventory

```
Stack counter = 5;
soulsand counter;    $$ Incrementa: counter = 6
magma counter;       $$ Decrementa: counter = 5
```

Operaciones básicas sobre caracteres

isEngraved, isInscribed, etchUp, etchDown

Notch Engine proporciona un conjunto de operaciones específicas para manipular caracteres. **isEngraved** verifica si un carácter es una letra, similar a **isAlpha**. **isInscribed** verifica si es un dígito, similar a **isDigit**. **etchUp** convierte un carácter a mayúscula, mientras que **etchDown** lo convierte a minúscula.

Ejemplo:

Inventory

```
Rune letter = 'a';
Torch isLetter = isEngraved(letter);    $$ Verifica si es letra: On
Torch isDigit = isInscribed(letter);    $$ Verifica si es dígito: Off
Rune upper = etchUp(letter);            $$ Convierte a mayúscula: 'A'
Rune lower = etchDown('A');             $$ Convierte a minúscula: 'a'
```

Operaciones lógicas solicitadas

and, or, not, xor

Las operaciones lógicas en Notch Engine se expresan mediante palabras en lugar de símbolos. Esto incluye **and** (conjunción), **or** (disyunción), **not** (negación) y **xor** (disyunción exclusiva).

Ejemplo:

Inventory

```
Torch a = On;
Torch b = Off;
Torch result1 = a and b;    $$ Conjunción: Off
Torch result2 = a or b;     $$ Disyunción: On
Torch result3 = not a;      $$ Negación: Off
Torch result4 = a xor b;    $$ Disyunción exclusiva: On
```

Operaciones de Strings solicitadas

bind, #, from ##, except ##, seek

Notch Engine proporciona un conjunto de operaciones específicas para manipular cadenas de texto. **bind** concatena dos cadenas, **#** devuelve la longitud de una cadena, **from ##** extrae una subcadena, **except ##** elimina una subcadena, y **seek** busca una subcadena dentro de otra. Las operaciones ternarias utilizan doble **#** para evitar ambigüedad con la operación de longitud.

Ejemplo:

Inventory

```
Spider a = "Hello";
Spider b = "World";
Spider result1 = bind(a, b);    $$ Concatenación: "HelloWorld"
Stack length = #(a);           $$ Longitud: 5
Spider result2 = from a ## 1 ## 3;  $$ Subcadena desde índice 1,
                                   $$ longitud 3: "ell"
Spider result3 = except a ## 1 ## 3;  $$ Elimina subcadena: "Ho"
Stack position = seek(a, "ll");    $$ Busca subcadena: 2
```

Operaciones de conjuntos solicitadas

Palabras reservadas: add, drop, feed, map, biom, void

En el lenguaje Notch Engine, se definen palabras reservadas específicas para manipular conjuntos:

- **add**: Agrega un elemento a un conjunto.
- **drop**: Elimina un elemento de un conjunto.
- **feed**: Realiza la unión de dos conjuntos.
- **map**: Realiza la intersección entre dos conjuntos.
- **biom**: Verifica si un elemento pertenece a un conjunto.
- **void**: Verifica si un conjunto está vacío.

Ejemplo de uso:

Inventory

```
Chest a = {: 1, 2, 3 :};
Chest b = {: 3, 4, 5 :};
add(a, 4);                $$ Agrega elemento: {: 1, 2, 3, 4 :}
drop(a, 1);               $$ Elimina elemento: {: 2, 3, 4 :}
Chest c = feed(a, b);     $$ Unión: {: 2, 3, 4, 5 :}
Chest d = map(a, b);      $$ Intersección: {: 3, 4 :}
Torch belongs = biom(a, 3); $$ Pertenece: On
Torch empty = void(a);    $$ Verifica si está vacío: Off
```

Operaciones de archivos solicitadas

unlock, lock, make, gather, forge, tag

Notch Engine proporciona operaciones específicas para manejar archivos. **unlock** abre un archivo, **lock** lo cierra, **make** crea un nuevo archivo, **gather** lee datos del archivo, **forge** escribe datos en el archivo, y **tag** concatena archivos.

Ejemplo:

Inventory

```
Book logFile = {/ "log.txt", 'E' /};
unlock(logFile);                $$ Abre el archivo
forge(logFile, "Log entry");    $$ Escribe en el archivo
lock(logFile);                  $$ Cierra el archivo

Book dataFile = make({/ "data.txt", 'E' /}); $$ Crea nuevo archivo
Spider content = gather(dataFile);          $$ Lee contenido
tag(logFile, dataFile);                  $$ Concatena archivos
```

Operaciones de números flotantes

`:+, :-, :*, :%, ://`

Las operaciones aritméticas para números flotantes utilizan los mismos símbolos que las operaciones para enteros, pero con dos puntos (:) antes del símbolo para diferenciarlas. Incluyen suma (:+), resta (:-), multiplicación (:*), división (:// - ¡ojo con la división!) y módulo (:%).

Ejemplo:

Inventory

```
Ghast a = 3.5 :+ 2.5;    $$ Suma flotante: 6.0
Ghast b = 3.5 :- 2.5;    $$ Resta flotante: 1.0
Ghast c = 3.5 :* 2.0;    $$ Multiplicación flotante: 7.0
Ghast d = 3.5 :// 2.0;   $$ División flotante: 1.75
Ghast e = 3.5 :% 2.0;    $$ Módulo flotante: 1.5
```

Operaciones de comparación solicitadas

`<, >, <=, >=, is, isNot`

Notch Engine proporciona operadores de comparación que devuelven valores booleanos. Estos incluyen menor que (<), mayor que (>), menor o igual que (<=), mayor o igual que (>=), igual a (is) y distinto de (isNot).

Ejemplo:

Inventory

```
Stack a = 5;
Stack b = 10;
Torch result1 = a < b;    $$ Menor que: On
Torch result2 = a > b;    $$ Mayor que: Off
Torch result3 = a <= b;   $$ Menor o igual que: On
Torch result4 = a >= b;   $$ Mayor o igual que: Off
Torch result5 = a is b;   $$ Igual a: Off
Torch result6 = a isNot b; $$ Distinto de: On
```

1.8. Estructuras de Control

Manejo de Bloques de más de una instrucción

PolloCrudo ... PolloAsado

En Notch Engine, los bloques de código que contienen múltiples instrucciones se delimitan con las palabras clave **PolloCrudo** y **PolloAsado**. Esta metáfora representa un proceso de transformación: al inicio está crudo (el código sin ejecutar) y al final del proceso está cocido o asado (el código ejecutado). Estos delimitadores permiten agrupar varias instrucciones para que sean tratadas como una sola unidad lógica.

Ejemplo:

```
PolloCrudo
    Stack counter = 1;
    dropperStack(counter);
    soulsand counter;
    dropperStack(counter);
PolloAsado
```

Instrucción while

repeater <cond> craft <instrucción>

La estructura de repetición **while** se implementa mediante la palabra clave **repeater**, haciendo referencia a cómo un repetidor de redstone en Minecraft envía señales de manera constante mientras recibe energía. Esta estructura ejecuta repetidamente una instrucción mientras la condición especificada sea verdadera. La palabra clave **craft** es parte integral de la sintaxis de esta instrucción.

Ejemplo:

```
Stack counter = 1;
repeater counter < 5 craft
    PolloCrudo
        dropperStack(counter);
        soulsand counter;
    PolloAsado
```

Instrucción if-then-else

target <cond> craft hit <inst> miss <inst>

La estructura condicional **if-then-else** se implementa mediante la palabra clave **target**, que evoca la idea del bloque objetivo de Minecraft que evalúa si se dio en el centro o no. La palabra **craft** forma parte de la sintaxis. La cláusula **hit** corresponde al bloque de código que se ejecuta si la condición es verdadera (se dio en el objetivo), mientras que la cláusula **miss** corresponde al bloque de código que se ejecuta si la condición es falsa (se falló

el objetivo). Es importante notar que `hit` y `miss` pueden aparecer en orden inverso, y ambos son opcionales, aunque al menos uno debe estar presente.

Ejemplo:

```
Stack score = 75;
target score >= 60 craft hit
    PolloCrudo
        dropperSpider("Pasaste la prueba");
    PolloAsado
miss
    PolloCrudo
        dropperSpider("Reprobaste la prueba");
    PolloAsado
```

Instrucción `switch`

`jukebox <condition> craft, disc <case>: <instrucción>, silence`

La estructura de selección múltiple `switch` se implementa mediante la palabra clave `jukebox`, evocando la caja de discos de Minecraft que permite seleccionar diferentes canciones. Cada caso se identifica con la palabra clave `disc` seguida del valor a comparar y dos puntos. La palabra clave `silence` reemplaza al `default` tradicional de otros lenguajes, y puede aparecer en cualquier posición. Es obligatorio usar los delimitadores `PolloCrudo` y `PolloAsado` para cada bloque de instrucciones.

Ejemplo:

```
Stack option = 2;
jukebox option craft
    disc 1:
        PolloCrudo
            dropperSpider("Opción 1 seleccionada");
        PolloAsado

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

    silence:
        PolloCrudo
            dropperSpider("Opción no válida");
        PolloAsado
```

Instrucción Repeat-until

spawner <instrucciones> exhausted <cond>;

La estructura de repetición **do-while** en Notch Engine se implementa con las palabras clave **spawner** y **exhausted**, haciendo referencia al generador de monstruos de Minecraft que continúa generando criaturas hasta que se cumpla una condición de parada. Esta estructura ejecuta un bloque de código al menos una vez y luego repite mientras la condición especificada sea falsa.

Ejemplo:

```
Stack attempts = 0;
Torch success = Off;
spawner
    PolloCrudo
        soulsand attempts;
        dropperSpider("Intento número: " + attempts);

        target attempts > 5 craft hit
            PolloCrudo
                success = On;
            PolloAsado
    PolloAsado
exhausted
    success;
```

Instrucción For

walk VAR set <exp> to <exp> step <exp> craft <instrucción>

La estructura de repetición **for** se implementa mediante la palabra clave **walk**, que evoca la idea de caminar o recorrer en Minecraft. Esta estructura permite iterar desde un valor inicial hasta un valor final con un incremento específico. La cláusula **step** define el incremento y es opcional (por defecto es 1).

Ejemplo:

```
walk i set 1 to 5 craft
    PolloCrudo
        dropperSpider("Iteración: " + i);
    PolloAsado
```

```
walk j set 10 to 0 step -2 craft
  PolloCrudo
    dropperSpider("Cuenta regresiva: " + j);
  PolloAsado
```

Instrucción With

with <Referencia a Record> **craft** <instrucción>

La estructura **with** se implementa mediante la palabra clave **with**, haciendo un juego de palabras con "*with*" y refiriéndose al enemigo Wither de Minecraft. Esta estructura permite trabajar de manera simplificada con los campos de un registro (record), evitando la necesidad de acceder explícitamente a cada campo con el operador @.

Ejemplo:

```
Entity Player
  Spider name;
  Stack health;
  Stack level;
```

```
Entity Player steve;
steve@name = "Steve";
steve@health = 20;
steve@level = 1;
```

```
with steve craft
  PolloCrudo
    soulsand level;
    health = 20;
    dropperSpider(name + " está en el nivel " + level + " con salud
    " + health);
  PolloAsado
```

Instrucción break

creeper

La instrucción **break**, que permite salir prematuramente de un bucle, se implementa mediante la palabra clave **creeper**. Esta referencia es apropiada ya que los creepers en Minecraft son conocidos por interrumpir abruptamente lo que el jugador está haciendo.

Ejemplo:

```
walk i set 1 to 10 craft
PolloCrudo
    dropperSpider("Iteración: " + i);

    target i is 5 craft hit
        PolloCrudo
            dropperSpider("Interrumpiendo en 5");
            creeper;
        PolloAsado
PolloAsado
```

Instrucción continue

enderPearl

La instrucción `continue`, que permite saltar a la siguiente iteración de un bucle, se implementa mediante la palabra clave `enderPearl`. Esta elección tiene sentido ya que las Ender Pearls en Minecraft permiten teletransportarse, similar a cómo la instrucción `continue` "salta" por encima del resto del código del bucle.

Ejemplo:

```
walk i set 1 to 10 craft
PolloCrudo
    target i % 2 is 0 craft hit
        PolloCrudo
            enderPearl; $$ Salta los números pares
        PolloAsado

    dropperSpider("Número impar: " + i);
PolloAsado
```

Instrucción Halt

ragequit

La instrucción `halt`, que termina inmediatamente la ejecución del programa, se implementa mediante la palabra clave `ragequit`. Esta elección transmite la idea de terminar repentinamente la ejecución, como cuando un jugador abandona el juego por frustración o enojo.

Ejemplo:

```
dropperSpider("Iniciando programa");
```

```
target hopperTorch() craft hit
PolloCrudo
    dropperSpider("Error crítico detectado");
    ragequit; $$ Termina la ejecución del programa
PolloAsado

dropperSpider("Continuando ejecución normal");
```

1.9. Funciones y Procedimientos

Encabezado de funciones

Spell <id>(<parameters>) -> <tipo>

En Notch Engine, las funciones se declaran mediante la palabra clave **Spell**, refiriéndose a los hechizos o encantamientos en Minecraft. Una función es un bloque de código que procesa parámetros de entrada y devuelve un valor específico. El tipo de retorno se indica después de una flecha (->), especificando el tipo de dato que la función devolverá.

Ejemplo:

```
Spell calcuDamage(Stack :: level, weapon; Ghast ref critMultiplier) -> Stack
PolloCrudo
    Stack baseDamage = level * 2 + 5;

    target weapon is 1 craft hit
    PolloCrudo
        baseDamage = baseDamage * 2; $$ Espada de diamante
    PolloAsado

    critMultiplier = 1.5; $$ Modificamos el parámetro por referencia

    respawn baseDamage;
PolloAsado
```

Encabezado de procedimientos

Ritual <id>(<parameters>)

Los procedimientos en Notch Engine se declaran con la palabra clave **Ritual**, evocando la idea de un ritual o ceremonia en Minecraft. A diferencia de las funciones, los procedimientos no devuelven un valor explícito, sino que realizan una serie de acciones o efectos secundarios.

Ejemplo:

```
Ritual displayPlayerInfo(Spider :: name; Stack level, health)
PolloCrudo
    dropperSpider("== Información del Jugador ==");
    dropperSpider("Nombre: " + name);
    dropperSpider("Nivel: " + level);
    dropperSpider("Salud: " + health);
    dropperSpider("=====");
PolloAsado
```

Manejo de parámetros formales

(<type>:: <name>, <name>; <type> ref <name>; ...)

Notch Engine utiliza una sintaxis específica para la declaración de parámetros en funciones y procedimientos:

- :: separa el tipo de dato de la lista de nombres de parámetros
- , separa múltiples parámetros del mismo tipo
- ; separa grupos de parámetros de diferentes tipos
- ref indica que un parámetro se pasa por referencia en lugar de por valor

Los parámetros pasados por valor no pueden ser modificados dentro de la función, mientras que los pasados por referencia sí pueden ser modificados, y estos cambios afectarán a la variable original.

Ejemplo:

\$\$ Función con múltiples tipos de parámetros

```
Spell calculateStats(Stack :: baseStrength, baseAgility;
                    Ghast :: modifier;
                    Torch ref hasLeveledUp) -> Stack
PolloCrudo
    Stack totalStats = (baseStrength + baseAgility) * modifier;

    target totalStats > 100 craft hit
        PolloCrudo
            hasLeveledUp = On;  $$ Modificamos el parámetro por referencia
        PolloAsado

    respawn totalStats;
PolloAsado
```

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, estos se pasan entre paréntesis y separados por comas.

Para los procedimientos, se utiliza el nombre del procedimiento y los parámetros reales entre paréntesis.

Ejemplo:

Inventory

```
Stack playerLevel = 5;
Stack playerHealth = 20;
Spider playerName = "Steve";
Ghast damageMultiplier = 1.0;
Torch leveledUp = Off;
```

\$\$ Llamada a una función con parámetros reales

```
Stack damage = calculateDamage(playerLevel, 1, damageMultiplier);
```

\$\$ Llamada a un procedimiento con parámetros reales

```
displayPlayerInfo(playerName, playerLevel, playerHealth);
```

\$\$ Usando un parámetro por referencia

```
Stack stats = calculateStats(10, 8, 1.2, leveledUp);
```

\$\$ Verificando el cambio en el parámetro por referencia

```
target leveledUp craft hit
```

```
    PolloCrudo
```

```
        dropperSpider("¡El jugador ha subido de nivel!");
```

```
    PolloAsado
```

Instrucción return

respawn

La instrucción **return**, que devuelve un valor desde una función, se implementa mediante la palabra clave **respawn** en Notch Engine. Esta elección refleja la mecánica de *"reaparecer"* en Minecraft, simbolizando cómo el valor *"reaparece"* en el punto de llamada de la función.

Ejemplo:


```

Spell calculateExperience(Stack :: level, kills) -> Stack
PolloCrudo
    $$ Return temprano en caso de error
    target level <= 0 craft hit
        PolloCrudo
            dropperSpider("Error: El nivel debe ser mayor que 0");
            respawn 0;
        PolloAsado

    Stack baseXP = level * 100;
    Stack killBonus = kills * 25;
    Stack totalXP = baseXP + killBonus;

    $$ Return normal al final de la función
    respawn totalXP;
PolloAsado

$$ Uso de la función
Stack playerLevel = 5;
Stack playerKills = 10;
Stack xpReward = calculateExperience(playerLevel, playerKills);
dropperSpider("Experiencia obtenida: " + xpReward);

```

Es importante destacar que la instrucción **respawn** también puede utilizarse sin un valor de retorno en los procedimientos para finalizar la ejecución del procedimiento antes de llegar al final del bloque.

```

Ritual checkPermissions(Stack :: playerRank)
PolloCrudo
    target playerRank < 3 craft hit
        PolloCrudo
            dropperSpider("Acceso denegado: Rango insuficiente");
            respawn;  $$ Termina el procedimiento temprano
        PolloAsado

    dropperSpider("Acceso concedido");
    $$ Continúa la ejecución...
PolloAsado

```

1.10. Elementos Auxiliares

Operación de size of

chunk <exp> o <tipo>

En Notch Engine, la operación para determinar el tamaño de una estructura de datos o un tipo se implementa mediante la palabra clave **chunk**. Esta operación recibe una expresión válida o un tipo y retorna un entero con la cantidad de bytes que ocupa en memoria, similar a cómo los chunks en Minecraft representan unidades de tamaño del mundo.

Ejemplos:

\$\$ Obtener el tamaño de una variable

```
Spider name = "Steve";
```

Stack nameSize = chunk name; \$\$ Retorna la cantidad de bytes que ocupa

\$\$ Obtener el tamaño de un tipo

```
Stack stackSize = chunk Stack;    $$ Retorna el tamaño en bytes del tipo Stack
```

\$\$ Obtener el tamaño de un arreglo

```
Shelf[10] inventory;
```

Stack arraySize = chunk inventory; \$\$ Retorna el tamaño total del arreglo

\$\$ Obtener el tamaño de una estructura

```
Entity Player
```

```
    Spider name;
```

```
    Stack health;
```

```
    Stack level;
```

Stack entitySize = chunk Player; \$\$ Retorna el tamaño de la estructura

Sistema de coerción de tipos

<exp> >> <tipo>

Notch Engine proporciona un operador de coerción de tipos mediante el símbolo **>>**. Este operador permite interpretar el resultado de una expresión como si fuera de otro tipo, sin convertirlo completamente. Es importante destacar que la coerción es diferente de la conversión automática de tipos, ya que no cambia realmente el valor, solo la forma en que se interpreta.

Ejemplos:

\$\$ Coerción de flotante a entero

```

Ghast pi = 3.14159;
Stack intPi = pi >> Stack;  $$ Se interpreta pi como Stack
                             $$ (truncando la parte decimal)

$$ Coherción de entero a booleano
Stack value = 42;
Torch boolValue = value >> Torch;  $$ Cualquier valor distinto de 0
                                     $$ se interpreta como On

$$ Coherción para acceso a bytes
Stack packedData = 0x12345678;
Rune firstByte = packedData >> Rune;  $$ Extrae el primer byte
                                       $$ como un carácter

```

Manejo de la entrada estándar

x = hopper<TipoBásico>()

Para la lectura de datos desde la entrada estándar, Notch Engine proporciona un conjunto de funciones predefinidas que comienzan con la palabra **hopper**, haciendo referencia a los bloques hopper (tolvas) de Minecraft que recogen ítems. Estas funciones no requieren argumentos y retornan un valor del tipo básico especificado.

Ejemplos:

```

$$ Leer un entero desde la entrada estándar
Stack level = hopperStack();

$$ Leer un flotante desde la entrada estándar
Ghast health = hopperGhast();

$$ Leer una cadena desde la entrada estándar
Spider name = hopperSpider();

$$ Leer un booleano desde la entrada estándar
Torch decision = hopperTorch();

$$ Leer un carácter desde la entrada estándar
Rune option = hopperRune();

```

Manejo de la salida estándar

dropper<tipoBásico>(dato)

Para la escritura de datos en la salida estándar, Notch Engine proporciona un conjunto de funciones predefinidas que comienzan con la palabra `dropper`, haciendo referencia a los bloques `dropper` (dispensadores) de Minecraft que expulsan ítems. Estas funciones reciben un argumento del tipo básico especificado y lo muestran en la salida estándar.

Ejemplos:

```
$$ Mostrar un entero en la salida estándar
```

```
Stack level = 42;  
dropperStack(level);
```

```
$$ Mostrar un flotante en la salida estándar
```

```
Ghast pi = 3.14159;  
dropperGhast(pi);
```

```
$$ Mostrar una cadena en la salida estándar
```

```
Spider message = "¡Bienvenido a Notch Engine!";  
dropperSpider(message);
```

```
$$ Mostrar un booleano en la salida estándar
```

```
Torch gameOver = Off;  
dropperTorch(gameOver);
```

```
$$ Mostrar un carácter en la salida estándar
```

```
Rune grade = 'A';  
dropperRune(grade);
```

Terminador o separador de instrucciones - Instrucción nula

;

En Notch Engine, al igual que en muchos otros lenguajes de programación, el punto y coma (;) se utiliza como terminador o separador de instrucciones. Cada instrucción simple debe finalizar con un punto y coma, mientras que las instrucciones compuestas (aquellas que contienen bloques delimitados por `PolloCrudo` y `PolloAsado`) no requieren punto y coma.

También se puede utilizar el punto y coma solo para representar una instrucción nula o vacía, que no realiza ninguna acción.

Ejemplos:

```
$$ Uso del punto y coma como terminador de instrucciones
```

```
Stack counter = 0;  
soulsand counter;
```

```
dropperStack(counter);
```

\$\$ Instrucción nula

target condition craft hit ; \$\$ No hace nada si la condición es verdadera

\$\$ En un bucle for, se puede usar instrucción nula para inicialización

\$\$ o incremento

```
walk i set ; i < 10; soulsand i craft
```

```
    PolloCrudo
```

```
        dropperStack(i);
```

```
    PolloAsado
```

Todo programa se debe cerrar con un

worldSave

Cada programa en Notch Engine debe terminar con la palabra clave **worldSave**, que simboliza el acto de guardar el mundo en Minecraft. Esta instrucción marca el final del programa y asegura que todos los recursos se liberan correctamente.

Ejemplo:

```
WorldName SimpleProgram:
```

```
Inventory
```

```
    Stack counter = 10;
```

```
    repeater counter > 0 craft
```

```
        PolloCrudo
```

```
            dropperStack(counter);
```

```
            magma counter;
```

```
        PolloAsado
```

```
SpawnPoint
```

```
    dropperSpider("Programa terminado");
```

```
worldSave
```

Comentario de Bloque

```
$* comentario *$
```

Los comentarios de bloque en Notch Engine se delimitan con `$*` al inicio y `*$` al final. Estos comentarios pueden abarcar múltiples líneas y son ignorados por el compilador. El símbolo del dólar (\$) hace referencia al dinero recaudado por Minecraft, uno de los juegos más exitosos de la historia.

Ejemplo:

```
$*
Este es un comentario de bloque en Notch Engine.
Puede abarcar múltiples líneas y es útil para documentación
extensa o para explicar secciones complejas de código.
```

El compilador ignora completamente este texto.

```
*$
```

```
SpawnPoint
    dropperSpider("Hola, mundo!");
```

```
worldSave
```

Comentario de Línea

\$\$ comentario

Los comentarios de línea en Notch Engine comienzan con `$$` y continúan hasta el final de la línea. Son útiles para explicaciones breves o notas rápidas. Al igual que los comentarios de bloque, el símbolo del dólar (\$) hace referencia al éxito financiero de Minecraft.

Ejemplo:

```
Inventory
    $$ Inicialización de variables
    Stack health = 20;  $$ Salud máxima del jugador
    Stack level = 1;    $$ Nivel inicial

    $$ Calcular el poder de ataque basado en el nivel
    Stack attackPower = level * 5;  $$ 5 puntos de daño por nivel
```

```
SpawnPoint
    dropperSpider(";Comienza la aventura!");  $$ Mensaje de bienvenida
```

```
worldSave
```

1.11. Listado de Palabras Reservadas

Las palabras reservadas son identificadores especiales que tienen un significado predefinido en el lenguaje y no pueden ser utilizados con otro propósito. A continuación se presenta el listado completo de palabras reservadas del lenguaje Notch Engine, organizadas por categorías.

Elementos de la Estructura del Programa

WorldName	Bedrock	ResourcePack	Inventory	Recipe
CraftingTable	SpawnPoint	Obsidian	Anvil	worldSave

Tipos de Datos

Stack	Rune	Spider	Torch	Chest
Book	Ghast	Shelf	Entity	ref

Literales Booleanas

On Off

Delimitadores de Bloques

PolloCrudo PolloAsado

Control de Flujo

repeater	craft	target	hit	miss
jukebox	disc	silence	spawner	exhausted
walk	set	to	step	wither
creeper	enderPearl	ragequit		

Funciones y Procedimientos

Spell Ritual respawn

Operadores de Caracteres

isEngraved isInscribed etchUp etchDown

Operadores Lógicos

and or not xor

Operadores de Strings

bind # from except seek

Operadores de Conjuntos

add drop feed map
biom kill

Operadores de Archivos

unlock lock make gather forge tag

Operadores de Comparación

is isNot

Funciones de Entrada/Salida

hopperStack	hopperRune	hopperSpider	hopperTorch
dropperStack	dropperRune	dropperSpider	dropperTorch
hopperChest	hopperGhast	dropperChest	dropperGhast

Otros Operadores

chunk soulsand magma

Capítulo 2

Gramatica del Lenguaje

2.1. 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

2.2. 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

```

2.3. 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

2.4. 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 ($\{\langle id \rangle: \langle value \rangle, \langle id \rangle: \langle value \rangle, \dots\}$)

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: 0n  
};
```

```
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

2.5. 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

```

2.6. 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

2.7. 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 (spawner <instrucciones> exhausted <cond>)

La instrucción **do-while** en Notch Engine se representa mediante las palabras clave **spawner** 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> ::= "spawner" <bloque> "exhausted"
                           <expresion-booleana> ";"
<instruccion-do-while> ::= "spawner" <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 (with <Referencia a Record> craft <instrucción>)

La instrucción **with** en Notch Engine se representa mediante la palabra clave **with**, haciendo un juego de palabras con "with" 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

```

2.8. 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.^{en} 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

```

2.9. 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 3

Algoritmos de Conversión

3.1. Introducción

Esta sección documenta los algoritmos utilizados para convertir valores entre los diferentes tipos de datos del lenguaje. Para cada tipo de dato origen, se detalla el proceso de conversión hacia todos los demás tipos posibles, incluyendo reglas de manejo de casos especiales, restricciones y ejemplos.

3.2. Tipos de Datos Básicos

El lenguaje soporta los siguientes tipos de datos básicos:

- **Stack:** Representa números enteros (ej: 42, -15).
- **Ghast:** Representa números con punto decimal (ej: 3.14, -0.5).
- **Rune:** Representa un único carácter (ej: 'a', '7', '\$').
- **Spider:** Representa una secuencia de caracteres (ej: "Hola mundo").
- **Torch:** Representa valores de verdad (On, Off).
- **Creativo:** Representa un valor de probabilidad entre 0 y 1.

De \ A	Stack	Ghast	Rune	Spider	Torch	Creativo
Stack	-					
Ghast		-				
Rune			-			
Spider				-		
Torch					-	
Creativo						-

Cuadro 3.1: Matriz de conversiones posibles entre tipos

3.3. Matriz de Conversiones

3.4. Conversiones desde Stack

Stack a Ghast

La conversión de un valor entero a flotante mantiene el valor numérico exacto y agrega una representación decimal. Esta conversión es directa y sin pérdida de precisión: el valor entero se convierte en la parte entera del número flotante, y se agrega un punto decimal seguido de ceros. **Proceso:**

1. Tomar el valor entero original sin modificación como la parte entera del flotante.
2. Establecer la parte decimal como cero.
3. Combinar ambas partes con un punto decimal.

Ejemplos:

- $42 \rightarrow 42,0$
- $-15 \rightarrow -15,0$
- $0 \rightarrow 0,0$

Stack a Rune

Esta conversión extrae el primer dígito significativo del valor entero y lo convierte en su representación como carácter. Para números de varios dígitos, solo se considera el dígito más significativo (el de la izquierda). **Proceso:**

1. Si el número es negativo, se trabaja con su valor absoluto.
2. Si el número tiene más de un dígito, se divide repetidamente por 10 hasta obtener un solo dígito.
3. El dígito resultante se convierte a su representación como carácter.

Ejemplos:

- $42 \rightarrow '4'$ (el primer dígito es 4)
- $-15 \rightarrow '1'$ (ignorando el signo negativo, el primer dígito es 1)
- $7 \rightarrow '7'$ (un solo dígito)

Casos especiales: El cero se convierte al carácter '0'.

Stack a Spider

La conversión a string transforma el valor entero en su representación textual, manteniendo el signo si es negativo. Esta conversión preserva el valor exacto del número. **Proceso:**

1. Si el número es negativo, se incluye el signo menos al inicio de la cadena.
2. Se extraen los dígitos del número uno por uno, del menos significativo al más significativo (de derecha a izquierda).
3. Se invierten los dígitos para formar la representación correcta.
4. Para el caso especial del cero, se retorna la cadena "0".

Ejemplos:

- $42 \rightarrow "42"$
- $-15 \rightarrow "15"$
- $0 \rightarrow "0"$

Stack a Torch

La conversión de entero a booleano sigue la convención de que cero representa falso y cualquier otro valor representa verdadero. **Proceso:**

1. Si el valor entero es exactamente 0, se convierte a Off.
2. Para cualquier otro valor (positivo o negativo), se convierte a On.

Ejemplos:

- $0 \rightarrow \text{Off}$
- $42 \rightarrow \text{On}$
- $-15 \rightarrow \text{On}$

Stack a Creativo

La conversión a nuestro tipo creativo (probabilidad) extrae el primer dígito significativo del entero y lo utiliza para crear un valor de probabilidad entre 0 y 1, con formato 0.X. **Proceso:**

1. Si el número es negativo, se trabaja con su valor absoluto.
2. Se extrae el primer dígito significativo mediante divisiones sucesivas por 10.
3. Se forma un valor de probabilidad con formato 0.X donde X es el dígito extraído.

Ejemplos:

- $42 \rightarrow 0,4$ (el primer dígito es 4)
- $-15 \rightarrow 0,1$ (ignorando el signo negativo, el primer dígito es 1)
- $7 \rightarrow 0,7$ (un solo dígito)
- $0 \rightarrow 0,0$ (caso especial para el cero)

3.5. Conversiones desde Ghast

Ghast a Stack

La conversión de un valor flotante a entero descarta la parte decimal, conservando únicamente la parte entera. Este proceso se conoce como truncamiento (no redondeo).

Proceso:

1. Se identifica la parte entera del número flotante (los dígitos antes del punto decimal).
2. Se descarta completamente la parte decimal (los dígitos después del punto).
3. Se mantiene el signo del número original.

Ejemplos:

- $3,14 \rightarrow 3$ (se descarta .14)
- $-2,7 \rightarrow -2$ (se descarta .7, manteniendo el signo negativo)
- $0,9 \rightarrow 0$ (aunque cercano a 1, se trunca a 0)

Casos especiales: Esta conversión puede resultar en pérdida de información cuando la parte decimal es distinta de cero.

Ghast a Rune

Esta conversión extrae el primer dígito significativo de la parte entera del número flotante y lo representa como un carácter.

Proceso:

1. Se obtiene el valor absoluto de la parte entera del número flotante.
2. Si la parte entera tiene más de un dígito, se divide repetidamente por 10 hasta obtener un solo dígito.
3. El dígito resultante se convierte a su representación como carácter.

Ejemplos:

- $3,14 \rightarrow '3'$ (el primer dígito de la parte entera es 3)
- $-2,7 \rightarrow '2'$ (ignorando el signo negativo, el primer dígito es 2)
- $0,9 \rightarrow '0'$ (la parte entera es 0)

Ghast a Spider

La conversión a string transforma el valor flotante en su representación textual, incluyendo el punto decimal y manteniendo el signo si es negativo.

Proceso:

1. Si el número es negativo, se incluye el signo menos al inicio de la cadena.
2. Se convierte la parte entera a su representación textual.
3. Se añade el punto decimal.
4. Se convierte la parte decimal a su representación textual, conservando los ceros a la derecha cuando sea necesario.

Ejemplos:

- $3,14 \rightarrow "3.14"$
- $-2,7 \rightarrow 2.70$ (se presentan dos decimales)
- $0,9 \rightarrow "0.90"$ (se presentan dos decimales)

Casos especiales: Esta implementación fija el número de decimales a mostrar en dos, independientemente de la precisión original del número.

Ghast a Torch

La conversión de flotante a booleano sigue la convención de que cero representa falso y cualquier otro valor representa verdadero.

Proceso:

1. Si el valor flotante es exactamente 0.0 (tanto la parte entera como decimal son cero), se convierte a Off.
2. Para cualquier otro valor (positivo, negativo, o con parte decimal), se convierte a On.

Ejemplos:

- $0,0 \rightarrow \text{Off}$
- $3,14 \rightarrow \text{On}$
- $-2,7 \rightarrow \text{On}$
- $0,001 \rightarrow \text{On}$ (aunque próximo a cero, no es exactamente cero)

Ghast a Creativo

La conversión a nuestro tipo creativo (probabilidad) extrae el primer dígito significativo del número y lo utiliza para crear un valor de probabilidad entre 0 y 1.

Proceso:

1. Se obtiene el valor absoluto del número flotante.
2. Si la parte entera es distinta de cero, se extrae su primer dígito significativo.
3. Si la parte entera es cero, se extrae el primer dígito significativo de la parte decimal.
4. Se forma un valor de probabilidad con formato 0.X donde X es el dígito extraído.

Ejemplos:

- $3,14 \rightarrow 0,3$ (el primer dígito de la parte entera es 3)
- $-2,7 \rightarrow 0,2$ (ignorando el signo negativo, el primer dígito es 2)
- $0,95 \rightarrow 0,9$ (la parte entera es 0, el primer dígito decimal es 9)

3.6. Conversiones desde Torch

Torch a Stack

La conversión de un valor booleano a entero sigue la convención estándar donde On se representa como 1 y Off como 0. **Proceso:**

1. Si el valor booleano es On, se convierte al entero 1.
2. Si el valor booleano es Off, se convierte al entero 0.

Ejemplos:

- $\text{On} \rightarrow 1$
- $\text{Off} \rightarrow 0$

Torch a Ghast

La conversión de un valor booleano a flotante sigue la misma convención que a entero, pero representando los valores como 1.0 y 0.0. **Proceso:**

1. Si el valor booleano es On, se convierte al flotante 1.0.
2. Si el valor booleano es Off, se convierte al flotante 0.0.

Ejemplos:

- On \rightarrow 1,0
- Off \rightarrow 0,0

Torch a Rune

La conversión a carácter representa los valores booleanos como los caracteres '1' para verdadero y '0' para falso. **Proceso:**

1. Si el valor booleano es On, se convierte al carácter '1'.
2. Si el valor booleano es Off, se convierte al carácter '0'.

Ejemplos:

- On \rightarrow '1'
- Off \rightarrow '0'

Torch a Spider

La conversión a string representa los valores booleanos como las cadenas de texto *On* y *Off*. **Proceso:**

1. Si el valor booleano es On, se convierte a la cadena "On".
2. Si el valor booleano es Off, se convierte a la cadena "Off".

Ejemplos:

- On \rightarrow "On"
- Off \rightarrow "Off"

Casos especiales: Las cadenas respetan el formato original de los literales booleanos del lenguaje.

Torch a Creativo

La conversión a nuestro tipo creativo (probabilidad) representa On como la probabilidad máxima y Off como la probabilidad mínima. **Proceso:**

1. Si el valor booleano es On, se convierte a la probabilidad 1.0 (100 %).
2. Si el valor booleano es Off, se convierte a la probabilidad 0.0 (0 %).

Ejemplos:

- On \rightarrow 1,0 (representando alta probabilidad)
- Off \rightarrow 0,0 (representando probabilidad nula)

b

3.7. Conversiones desde Rune

Rune a Stack

La conversión de un carácter a entero retorna el código ASCII del carácter, representándolo como un valor numérico. **Proceso:**

1. Se identifica el código ASCII asociado al carácter.
2. Este valor numérico se retorna como entero.

Ejemplos:

- 'A' \rightarrow 65 (código ASCII de la letra A mayúscula)
- '5' \rightarrow 53 (código ASCII del dígito 5, no el valor 5)
- '\$' \rightarrow 36 (código ASCII del símbolo de dólar)

Casos especiales: Para los caracteres que representan dígitos numéricos ('0' a '9'), debe notarse que se retorna su código ASCII (48 a 57) y no el valor del dígito que representan.

Rune a Ghast

La conversión de un carácter a flotante retorna el código ASCII del carácter como un número con punto decimal, donde la parte decimal es cero.

Proceso:

1. Se obtiene el código ASCII del carácter.
2. Se convierte este valor a flotante añadiendo un punto decimal y cero como parte decimal.

Ejemplos:

- 'A' \rightarrow 65,0
- '5' \rightarrow 53,0
- '\$' \rightarrow 36,0

Rune a Torch

La conversión a booleano sigue la convención de que solo el carácter nulo (con código ASCII 0) representa falso, mientras que cualquier otro carácter representa verdadero. **Proceso:**

1. Si el carácter tiene un código ASCII de 0 (carácter nulo), se convierte a Off.
2. Para cualquier otro carácter, se convierte a On.

Ejemplos:

- 'A' \rightarrow On
- " \rightarrow Off (carácter nulo)
- '0' \rightarrow On (el carácter '0' tiene código ASCII 48, no 0)

Casos especiales: Es importante distinguir entre el carácter nulo " (con código ASCII 0) y el carácter '0' (con código ASCII 48). Solo el carácter nulo se convierte a Off.

Rune a Spider

La conversión a string crea una cadena de texto de longitud 1 que contiene únicamente el carácter convertido. **Proceso:**

1. Se crea una cadena de texto vacía.
2. Se añade el carácter a la cadena.
3. Se retorna la cadena resultante de longitud 1.

Ejemplos:

- $'A' \rightarrow "A"$
- $'5' \rightarrow "5"$
- $'\$' \rightarrow "\$"$

Rune a Creativo

La conversión a nuestro tipo creativo (probabilidad) asigna un valor binario basado en si el carácter es nulo o no. **Proceso:**

1. Si el carácter es nulo (código ASCII 0), se convierte a la probabilidad 0.0.
2. Para cualquier otro carácter, se convierte a la probabilidad 1.0.

Ejemplos:

- $'A' \rightarrow 1,0$
- $' ' \rightarrow 0,0$
- $'0' \rightarrow 1,0$

3.8. Conversiones desde Spider

Spider a Stack

La conversión de una cadena de texto a entero suma los valores ASCII de todos los caracteres en la cadena. **Proceso:**

1. Se inicializa un contador a cero.
2. Se recorren todos los caracteres de la cadena de texto.

3. Para cada carácter, se suma su valor ASCII al contador.
4. El resultado final es la suma total de los valores ASCII.

Ejemplos:

- "55" \rightarrow 106
- "ABC" \rightarrow 198
- "3,14" \rightarrow 198

Casos especiales: Si la cadena está vacía, se retorna 0.

Spider a Ghast

La conversión a flotante utiliza la suma ASCII de la cadena como parte entera y añade .00 como parte decimal. **Proceso:**

1. Se calcula la suma ASCII total igual que en la conversión a entero.
2. Se usa este valor como parte entera del flotante.
3. Se añade ",00" como parte decimal fija.

Ejemplos:

- "55" \rightarrow 106,00
- "ABC" \rightarrow 198,00
- "3.14" \rightarrow 198,00

Casos especiales: Si la cadena está vacía, se retorna 0.00.

Spider a Torch

La conversión a booleano compara la cadena (insensible a mayúsculas/-minúsculas) con valores predefinidos. **Proceso:**

1. Se compara la cadena con "On" (cualquier combinación de mayúsculas/minúsculas) o "1".
2. Si coincide con alguno de estos valores, se convierte a On.
3. Se compara la cadena con "Off" (cualquier combinación de mayúsculas/minúsculas) o "0".

4. Si coincide con alguno de estos valores, se convierte a Off.
5. Para cualquier otra cadena, se retorna Off por defecto.

Ejemplos:

- "On" → On
- "ON" → On (insensible a mayúsculas/minúsculas)
- "on" → On (insensible a mayúsculas/minúsculas)
- "1" → On
- "Off" → Off
- "0" → Off
- "Hola" → Off (no es uno de los valores reconocidos)

Casos especiales: La comparación es insensible a mayúsculas y minúsculas.

Spider a Rune

La conversión a carácter extrae el primer carácter de la cadena. **Proceso:**

1. Se toma el primer carácter de la cadena.
2. Si la cadena está vacía, se utiliza un espacio como carácter por defecto.

Ejemplos:

- "Hola" → 'H'
- "123" → '1'
- → ' ' (espacio)

Spider a Creativo

La conversión a nuestro tipo creativo (probabilidad) utiliza la suma de los códigos ASCII de todos los caracteres para generar un valor entre 0 y 1.

Proceso:

1. Se calcula la suma de los códigos ASCII de todos los caracteres en la cadena.

2. Se extrae el primer dígito significativo de esta suma.
3. Se forma un valor de probabilidad con formato 0.X donde X es el dígito extraído.

Ejemplos:

- "Hola \rightarrow 0,3 (si la suma ASCII fuera 388, el primer dígito es 3)
- "42 \rightarrow 0,1 (si la suma ASCII fuera 102, el primer dígito es 1)
- \rightarrow 0,0 (cadena vacía, suma ASCII = 0)

3.9. Conversiones desde Creativo

Creativo a Stack

La conversión de nuestro tipo creativo (probabilidad) a entero extrae la parte decimal y la representa como un número entero. **Proceso:**

1. Se extrae la parte decimal del valor de probabilidad.
2. Se convierte esta parte decimal a un valor entero (por ejemplo, 0.7 se convierte en 7).

Ejemplos:

- 0,7 \rightarrow 7 (la parte decimal es 7)
- 0,0 \rightarrow 0 (la parte decimal es 0)
- 0,314 \rightarrow 314 (la parte decimal completa)

Casos especiales: La parte entera (siempre 0) se ignora en esta conversión.

Creativo a Ghast

La conversión a flotante utiliza directamente el valor de probabilidad como valor flotante. **Proceso:**

1. El valor de probabilidad ya es un número flotante, así que se retorna sin modificación.

Ejemplos:

- 0,7 \rightarrow 0,7
- 0,0 \rightarrow 0,0
- 0,314 \rightarrow 0,314

Casos especiales: Esta conversión es directa y no presenta casos especiales.

Creativo a Torch

La conversión a booleano interpreta el valor de probabilidad comparándolo con un umbral de 0.5. **Proceso:**

1. Si el valor de probabilidad es mayor o igual a 0.5, se convierte a On.
2. Si el valor de probabilidad es menor que 0.5, se convierte a Off.

Ejemplos:

- $0,7 \rightarrow \text{On}$
- $0,5 \rightarrow \text{On}$
- $0,3 \rightarrow \text{Off}$
- $0,0 \rightarrow \text{Off}$

Casos especiales: El valor exacto de 0.5 se considera On.

Creativo a Rune

La conversión a carácter extrae el primer dígito decimal de la probabilidad y lo convierte en su representación como carácter. **Proceso:**

1. Se extrae el primer dígito decimal del valor de probabilidad.
2. Este dígito se convierte a su representación como carácter.

Ejemplos:

- $0,7 \rightarrow '7'$ (el primer dígito decimal es 7)
- $0,0 \rightarrow '0'$ (el primer dígito decimal es 0)
- $0,3 \rightarrow '3'$ (el primer dígito decimal es 3)

Creativo a Spider

La conversión a string convierte el valor de probabilidad en su representación textual. **Proceso:**

1. Se convierte el valor numérico completo (parte entera y parte decimal) a su representación como cadena de texto.

Ejemplos:

- $0,7 \rightarrow "0,7"$
- $0,0 \rightarrow "0,0"$
- $0,314 \rightarrow "0,314"$

Capítulo 4

Listado de Errores Léxicos

4.1. Introducción

Esta sección presenta un listado numerado de los posibles errores léxicos que el analizador léxico (scanner) de Notch Engine puede detectar durante el procesamiento del código fuente. Cada error está identificado con un código único que facilita su referencia en mensajes de error y documentación. Los errores están organizados por categorías según los elementos del lenguaje.

4.2. Tabla de Errores Léxicos

Errores de caracteres generales

- E1** Carácter no reconocido: El carácter no pertenece al alfabeto del lenguaje. Ejemplo: Caracteres especiales no definidos en Notch Engine.
- E2** Carácter Unicode no soportado: Se utilizó un carácter fuera del conjunto ASCII soportado. Ejemplo: Caracteres de otros idiomas o símbolos especiales no incluidos.

Errores de strings y caracteres (Rune y Spider)

- E3** String sin cerrar: Un literal de string no tiene la comilla doble de cierre. Ejemplo: Creeper en el mundo
- E4** Carácter sin cerrar: Un literal de carácter (Rune) no tiene la comilla simple de cierre. Ejemplo: 'A
- E5** Literal de carácter vacío: Se definió un literal de carácter sin contenido. Ejemplo: ”

E6 Secuencia de escape inválida: Se utilizó una secuencia de escape no reconocida. Ejemplo: "\z"

E7 Múltiples caracteres en literal de carácter: Se colocó más de un carácter en un Rune. Ejemplo: 'abc'

Errores de comentarios

E8 Comentario de bloque sin cerrar: Un comentario de bloque no tiene el terminador `*.Ejemplo: $* Comentario sin cerrar`

Errores de números (Stack y Ghast)

E8 Múltiples puntos decimales: Un literal Ghast contiene más de un punto decimal. Ejemplo: 3.14.15

E9 Número mal formado: Un literal numérico tiene una estructura incorrecta. Ejemplo: 3.

E10 Operador flotante incompleto: Un operador para flotantes está mal formado. Ejemplo: :+

Errores específicos de Notch Engine

E11 Literal de conjunto mal formado: Un literal de conjunto (Chest) no tiene el formato correcto. Ejemplo: { : 1, 2,, 3 : }

E12 Literal de archivo mal formado: Un literal de archivo (Book) tiene un formato incorrecto. Ejemplo: { / "archivo.txt" , 'X' / }

E13 Literal de registro mal formado: Un literal de registro (Entity) tiene un formato incorrecto. Ejemplo: { campo1: , campo2: 5 }

E14 Literal de arreglo mal formado: Un literal de arreglo (Shelf) tiene un formato incorrecto. Ejemplo: [1, 2, ,3]

Errores de identificadores

E15 Identificador mal formado: Un identificador tiene una estructura incorrecta. Ejemplo: comienza con mayúscula o número.

E16 Identificador demasiado largo: Excede la longitud máxima permitida. Ejemplo: Más de 64 caracteres.

Errores de delimitadores

- E17** Delimitador PolloCrudo sin cerrar: Falta el correspondiente PolloAsado.
- E18** PolloAsado sin apertura: Se encontró un PolloAsado sin su correspondiente PolloCrudo.
- E19** Delimitadores de estructuras de control incompletos: Falta parte de la estructura como `craft`, `hit`, `miss`.

Errores de palabras reservadas

- E20** Palabra reservada mal escrita: Error de escritura. Ejemplo: `Recype` en lugar de `Recipe`.
- E21** Palabra reservada en contexto incorrecto: Palabra usada en un contexto inválido. Ejemplo: `WorldName` dentro de una función.

Errores de operadores

- E22** Operador de coerción incompleto: El operador `~` sin tipo de destino.
Operador de acceso incompleto: Operador `@` sin campo o `[]` sin índice.

Errores de buffer y archivo

- E22** Error de lectura de archivo: Problema al leer el archivo de entrada.
- E23** Fin de archivo inesperado: Final del archivo en medio de un token.
- E24** Buffer overflow: Se excedió el tamaño del buffer para un token.

4.3. Recuperación de Errores Críticos

Siguiendo los requisitos del proyecto, el scanner de Notch Engine nunca detiene su ejecución completamente ante un error. Incluso ante condiciones críticas, se implementan estrategias de recuperación para continuar el análisis y reportar la mayor cantidad posible de errores.

Para errores particularmente severos, se implementa una recuperación avanzada en los siguientes casos:

1. Se detectan más de 5 errores en una misma línea.

2. Se acumulan más de 10 errores en 20 tokens consecutivos.
3. Se produce un error de buffer overflow.
4. Se detecta un error de lectura de archivo.

Ante estas condiciones, el analizador:

- Descarta tokens problemáticos hasta encontrar un delimitador confiable (por ejemplo, punto y coma, fin de línea, o delimitadores como `PolloAsado`).
- Registra todos los errores encontrados y emite un mensaje de advertencia especial.
- Restablece el estado interno del analizador para reanudar el procesamiento.
- Continúa el análisis desde el punto de recuperación identificado, generando una advertencia de posible omisión de errores en la sección afectada.

4.4. Recuperación de Error para Evitar Errores en Cascada

Para evitar la propagación de errores (errores en cascada), se han implementado técnicas específicas en diferentes partes del scanner:

- **Al procesar identificadores y literales:** Se consume la secuencia de caracteres hasta encontrar un delimitador o espacio en blanco, de manera que un error en un identificador no afecte a los tokens posteriores.
- **En el manejo de strings y comentarios:** Se asume un cierre automático (al final de la línea o del archivo) en caso de no encontrar el delimitador de cierre, lo que evita que el error se propague a lo largo del documento.
- **Para números y operadores mal formados:** Se consume la secuencia conflictiva hasta identificar un patrón válido, evitando que un error numérico arrastre errores en la interpretación de operadores adyacentes.

- **En delimitadores de bloques:** Se aplica una detección de cierre automático, lo que permite que si falta el delimitador de cierre, el analizador asuma el final de la estructura y continúe con el análisis sin detenerse.

Estas estrategias aseguran que, aun en presencia de errores severos, el compilador sea capaz de detectar y reportar la mayor cantidad posible de errores en una sola ejecución, mejorando la experiencia del desarrollador.

Capítulo 5

Documentacion del Automata

5.1. Introducción

En este documento se presenta la documentación técnica del autómata diseñado para el lenguaje de programación "Notch Engine". El autómata fue desarrollado utilizando Mermaid.js (<https://www.mermaidchart.com/>), una herramienta que permite la creación de diagramas de manera sencilla y visualmente atractiva.

5.2. Consideraciones Técnicas

- **Restricción de Mermaid:** El compilador de Mermaid no admite el símbolo ":" en los nodos, por lo que en los diagramas este símbolo se representa como "dos puntos".
- **Modularización:** El autómata completo se dividió en submódulos especializados para mejorar la legibilidad y mantenimiento.
- **Flujo de procesamiento:** Cada submódulo procesa un tipo específico de token del lenguaje.

5.3. Diagrama Principal (Main)

5.3.1. Descripción

El módulo principal actúa como punto de entrada del autómata y como distribuidor hacia los submódulos especializados. Su función es analizar el primer carácter de cada token y redirigirlo al módulo correspondiente para su procesamiento completo.

5.3.2. Características

- Estado inicial único para todo el sistema
- No procesa tokens completos, solo realiza el enrutamiento
- Maneja la transición entre diferentes tipos de tokens
- Detecta caracteres no válidos y los dirige al manejador de errores

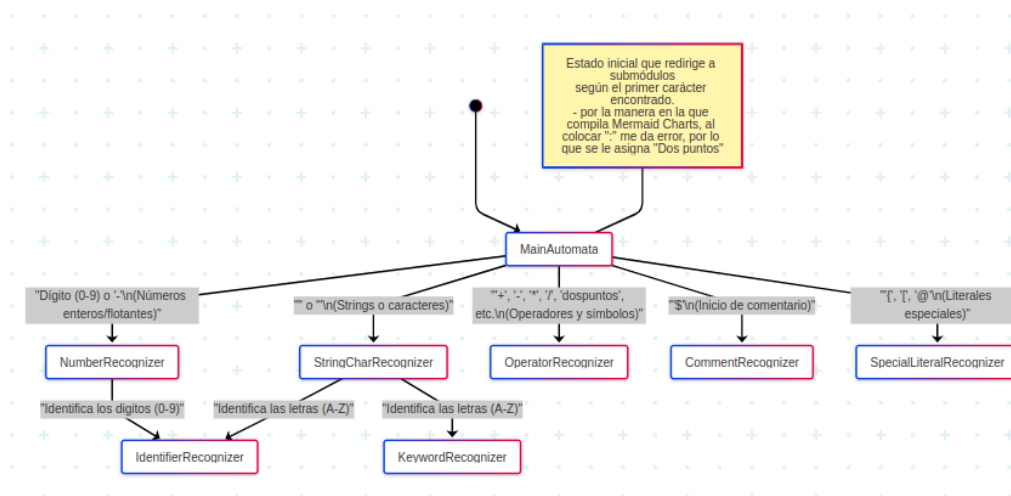


Figura 5.1: Automata del módulo principal del autómata

5.3.3. Lógica de Enrutamiento

El módulo principal del autómata funciona como un sistema de clasificación inicial que determina qué submódulo especializado procesará cada token basándose en su primer carácter. Esta lógica de enrutamiento es fundamental para el análisis léxico eficiente del código fuente. A continuación se detalla el proceso completo:

Mecanismo de Distribución

Cuando el autómata recibe un flujo de caracteres:

1. Examina el primer carácter no espaciado
2. Clasifica el carácter según categorías predefinidas
3. Redirige el procesamiento al submódulo correspondiente

4. Espera a que el submódulo complete el reconocimiento
5. Reinicia el proceso para el siguiente token

Tabla de Enrutamiento Detallada

Carácter Inicial	Módulo Destino	Ejemplos
Dígito (0-9) o '-'	NumberRecognizer	123, -3.14, 0xFF
Comilla doble (") o simple (')	StringCharRecognizer	"Hola", 'a', \n
Símbolo de dólar (\$)	CommentRecognizer	\$\$ comentario, \$* bloque *\$
Letra mayúscula (A-Z)	KeywordRecognizer	Bedrock, Entity, Torch
Letra minúscula (a-z)	IdentifierRecognizer	miVariable, contador1, x
Símbolos operadores	OperatorRecognizer	+, +=, :+, soulsand
Llaves, corchetes	SpecialLiteralRecognizer	{:1,2,3:}, [1,2,3], {/file.txt/}

Cuadro 5.1: Tabla completa de enrutamiento con ejemplos

Reglas Específicas por Módulo

- **Number Recognizer:**
 - Procesa literales numéricos enteros y flotantes
 - Ejemplos válidos: 42, -3.14
 - Ejemplos inválidos: 123abc, ..5
- **String/Char Recognizer:**
 - Maneja strings (delimitados por ") y caracteres (delimitados por ')
 - Admite secuencias de escape: \n, \t, \"
 - Ejemplo válido: Ruta\nC:\texto"
 - Error: 'string' (comillas simples para caracteres individuales)
- **Operator Recognizer:**
 - Reconoce operadores matemáticos, lógicos y especiales
 - Maneja versiones simples y compuestas: + vs +=
 - Operadores especiales: soulsand (++), magma (-)
 - Ejemplo: x :+ y (suma flotante)
- **Comment Recognizer:**

- Detecta comentarios de línea (`$$`) y de bloque (`$* *$`)
- Ignora completamente el contenido de los comentarios
- Ejemplo: `$$ Esto es un comentario`
- **Special Literal Recognizer:**
 - Procesa estructuras complejas como conjuntos, arreglos y registros
 - Ejemplo conjunto: `{'a', 'b', 1, 2}`
 - Ejemplo archivo: `{"data.txt", 'r' /}`
- **Identifier Recognizer:**
 - Maneja nombres definidos por el usuario (variables, funciones)
 - Reglas: comenzar con letra, puede contener dígitos, no palabras clave
 - Ejemplo válido: `contador1`
 - Error: `1variable` (no puede comenzar con dígito)
- **Keyword Recognizer:**
 - Identifica palabras reservadas exactas del lenguaje
 - Distingue mayúsculas/minúsculas según especificación
 - Ejemplo: `CraftingTable`, `worldSave`
 - Error: `craftingtable` (no coincide exactamente)

Este sistema de enrutamiento garantiza que cada token sea procesado por el submódulo más adecuado, optimizando el análisis léxico y facilitando la detección temprana de errores.

5.4. Reconocedor de Números

5.4.1. Descripción

Identifica y valida los literales numéricos del lenguaje, incluyendo enteros y flotantes.

5.4.2. Tipos Numéricos

- **Enteros (Stack):** Ej. 123, -5
- **Flotantes (Ghast):** Ej. 3.14, -0.5
- **Notación:** Admite números positivos y negativos

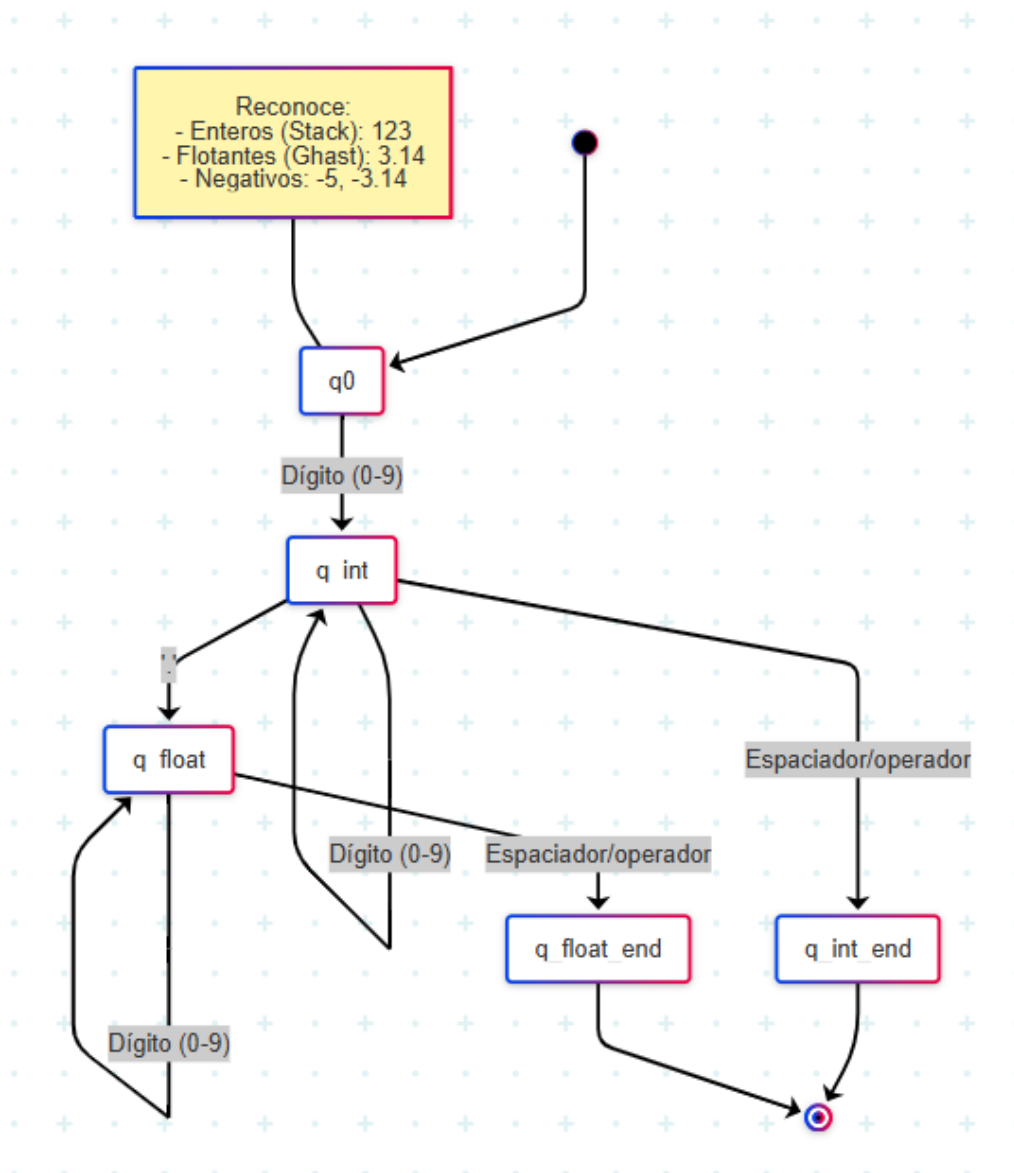


Figura 5.2: Diagrama del reconocedor de números

5.5. Reconocedor de Strings y Caracteres

5.5.1. Descripción

Procesa los literales de texto del lenguaje, incluyendo strings (Spider) y caracteres individuales (Rune).

5.5.2. Características

- **Strings:** Delimitados por comillas dobles ("...")
- **Caracteres:** Delimitados por comillas simples ('...')
- Admite secuencias de escape (ej. `\n`, `\t`)
- Validación de cierre de delimitadores

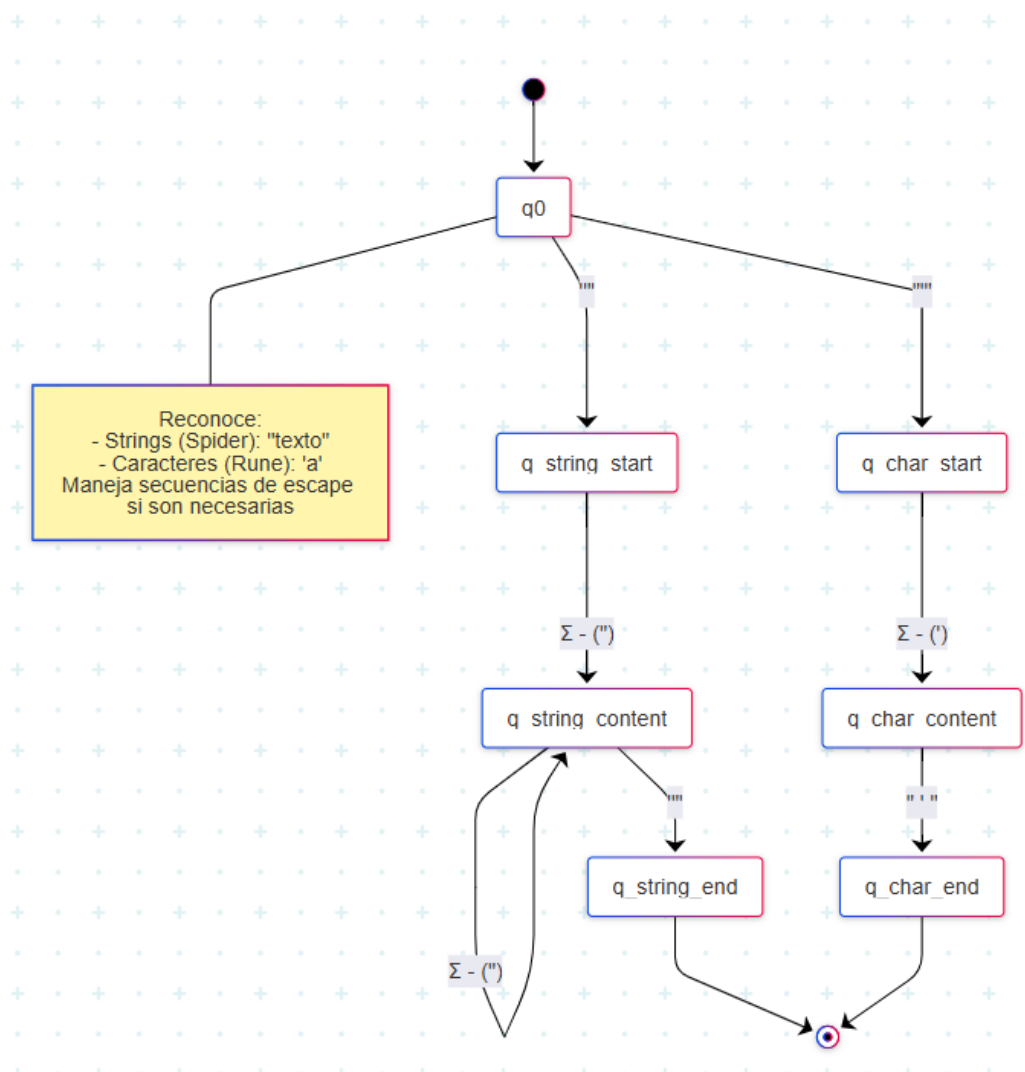


Figura 5.3: Diagrama del reconocedor de strings y caracteres

5.6. Reconocedor de Operadores y Símbolos

5.6.1. Descripción

Procesa los operadores del lenguaje, incluyendo los operadores tradicionales y los especiales de Notch Engine.

5.6.2. Categorías de Operadores

- **Aritméticos:** +, -, *, //, %, :+, :-, :*, ://, :%

- **Lógicos:** and, or, not, xor
- **Comparación:** <, >, <=, >=, *is*, *isNot*
- **Especiales:** soulsand (++) , magma (-)

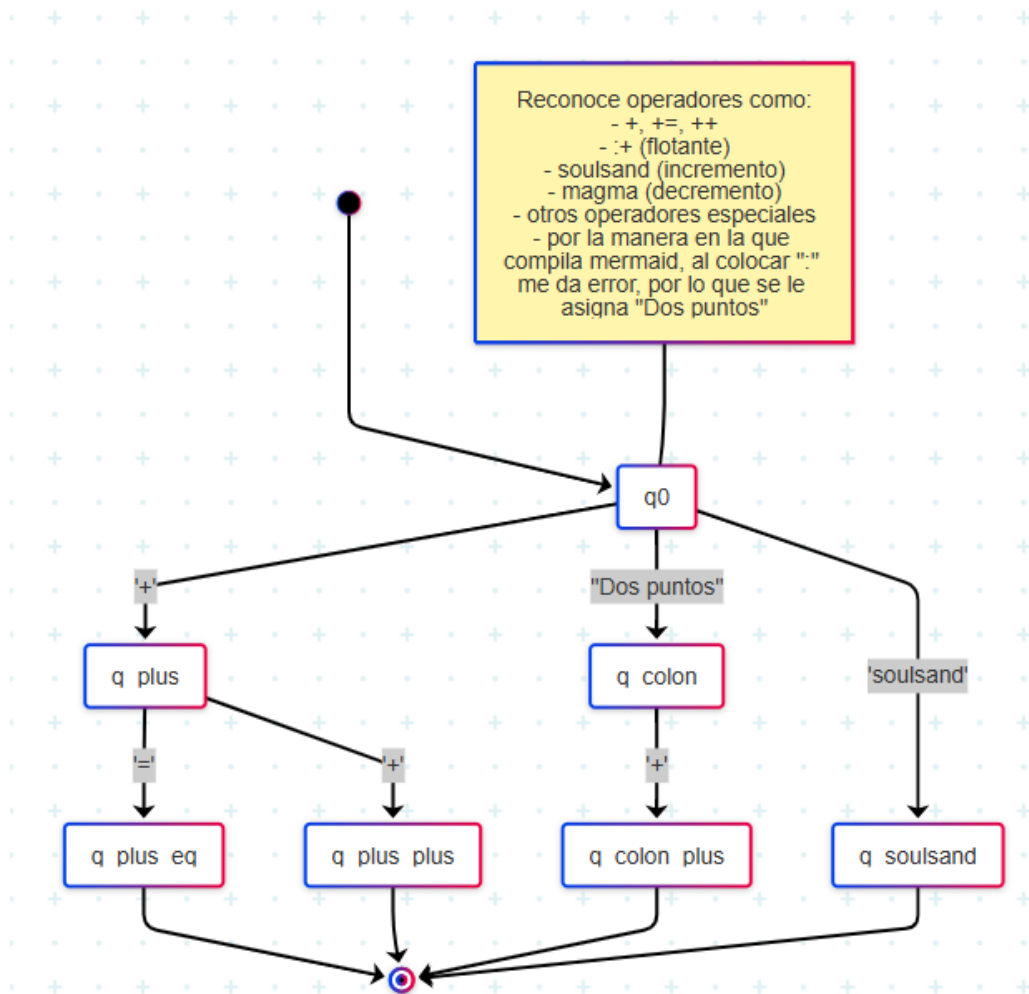


Figura 5.4: Diagrama del reconocedor de operadores y símbolos

5.7. Reconocedor de Comentarios

5.7.1. Descripción

Este submódulo se encarga de identificar y procesar los comentarios del lenguaje, que pueden ser de dos tipos: de línea (`$$`) y de bloque (`$* ... *$`).

5.7.2. Estados

- `q0`: Estado inicial
- `q_comment_line_start`: Detectado primer `$`
- `q_comment_line`: Contenido del comentario de línea
- `q_comment_block_start`: Detectado `$*`
- `q_comment_block`: Contenido del comentario de bloque
- `q_comment_block_aster`: Detectado `*` dentro de bloque
- `q_comment_block_end`: Comentario de bloque cerrado

- **Arreglos:** Formato [elemento1, elemento2]
- **Registros:** Formato campo: valor

5.9. Reconocedor de Identificadores

5.9.1. Descripción

Procesa los nombres de variables, funciones y otros identificadores definidos por el usuario, asegurando que cumplan con las reglas del lenguaje.

5.9.2. Reglas de Identificadores

- Deben comenzar con letra minúscula
- Pueden contener letras y dígitos
- No pueden coincidir con palabras reservadas
- Longitud máxima determinada por implementación

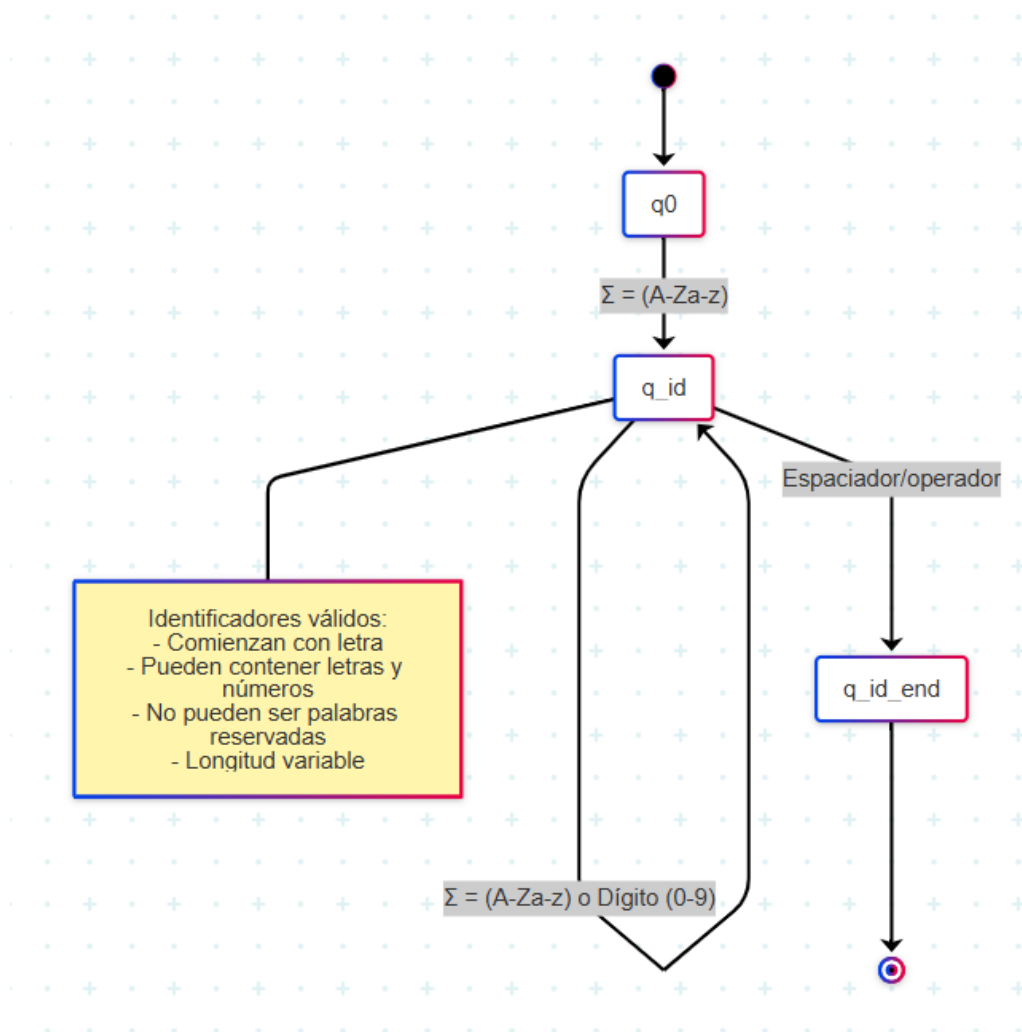


Figura 5.7: Diagrama del reconocedor de identificadores

5.10. Reconocedor de Palabras Reservadas

5.10.1. Propósito

El módulo de palabras reservadas identifica las palabras clave predefinidas del lenguaje "Notch Engine", diferenciándolas de identificadores regulares. Este componente es esencial para:

- Detectar términos con significado sintáctico especial
- Prevenir el uso de palabras clave como identificadores

- Habilitar el análisis gramatical preciso

5.10.2. Características Principales

- **Sensibilidad a mayúsculas:** Reconoce exactamente la capitalización definida (ej. `Bedrock` vs `bedrock`)
- **Validación completa:** Solo acepta palabras completas, no prefijos
- **Manejo de errores:** Detecta secuencias inválidas que parezcan palabras clave
- **Eficiencia:** Procesamiento en tiempo lineal $O(n)$

5.10.3. Palabras Implementadas

El sistema actual reconoce las siguientes palabras clave (como ejemplo demostrativo):

- `add` (operador)
- `and` (operador lógico)
- `Anvil` (declaración de tipos)
- `Bedrock` (sección de constantes)
- `biom` (operación de conjuntos)

5.10.4. Comportamiento

- **Transiciones válidas:** Sigue secuencias de caracteres que formen palabras reservadas
- **Estado de error:** Cualquier desviación de las secuencias esperadas
- **Aceptación:** Requiere delimitador después de la palabra completa (espacio, operador, etc.)

5.10.5. Restricciones

- No permite dígitos en palabras reservadas
- Rechaza sufijos o prefijos no válidos
- Distingue entre palabras clave y identificadores similares

5.10.6. Diagrama

A continuacion se muestra todo el automata en grupos de 5, esto para no sobrecargar cada imagen.

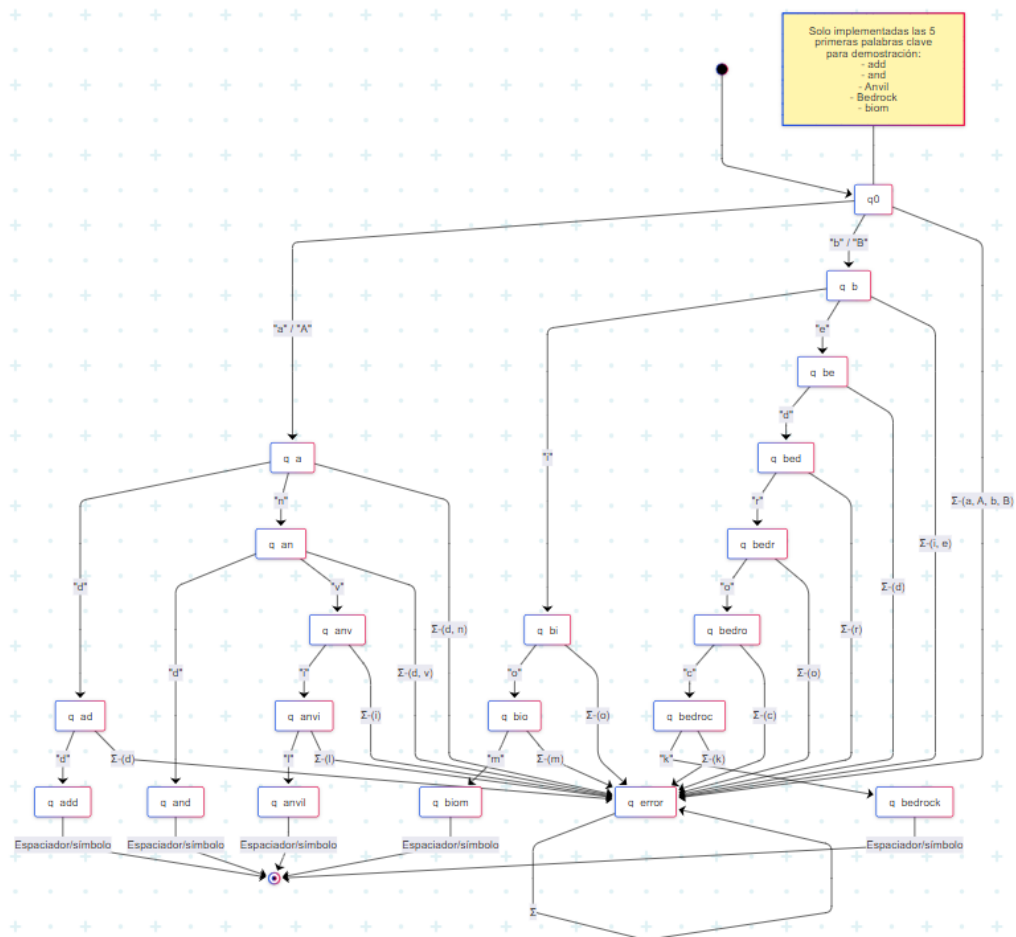


Figura 5.8: Diagrama del reconocedor de palabras reservadas de add-biom

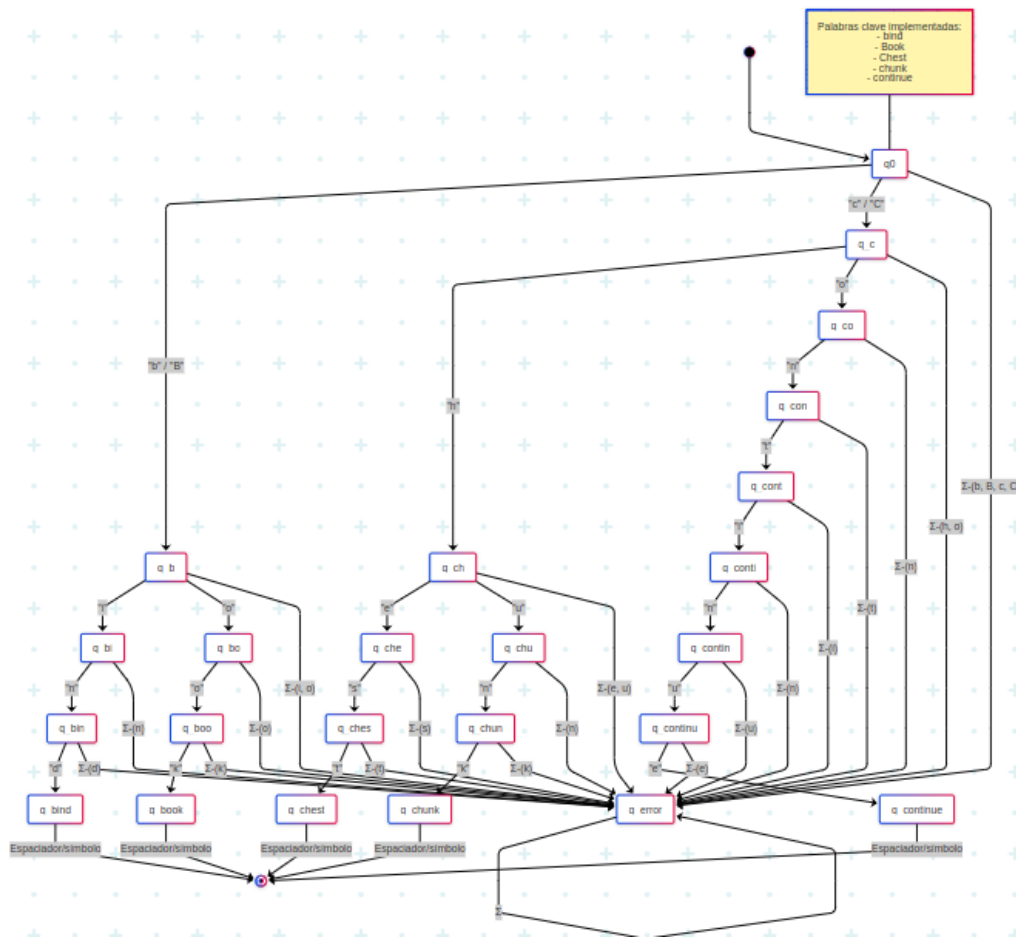


Figura 5.9: Diagrama del reconocedor de palabras reservadas de bind-continue

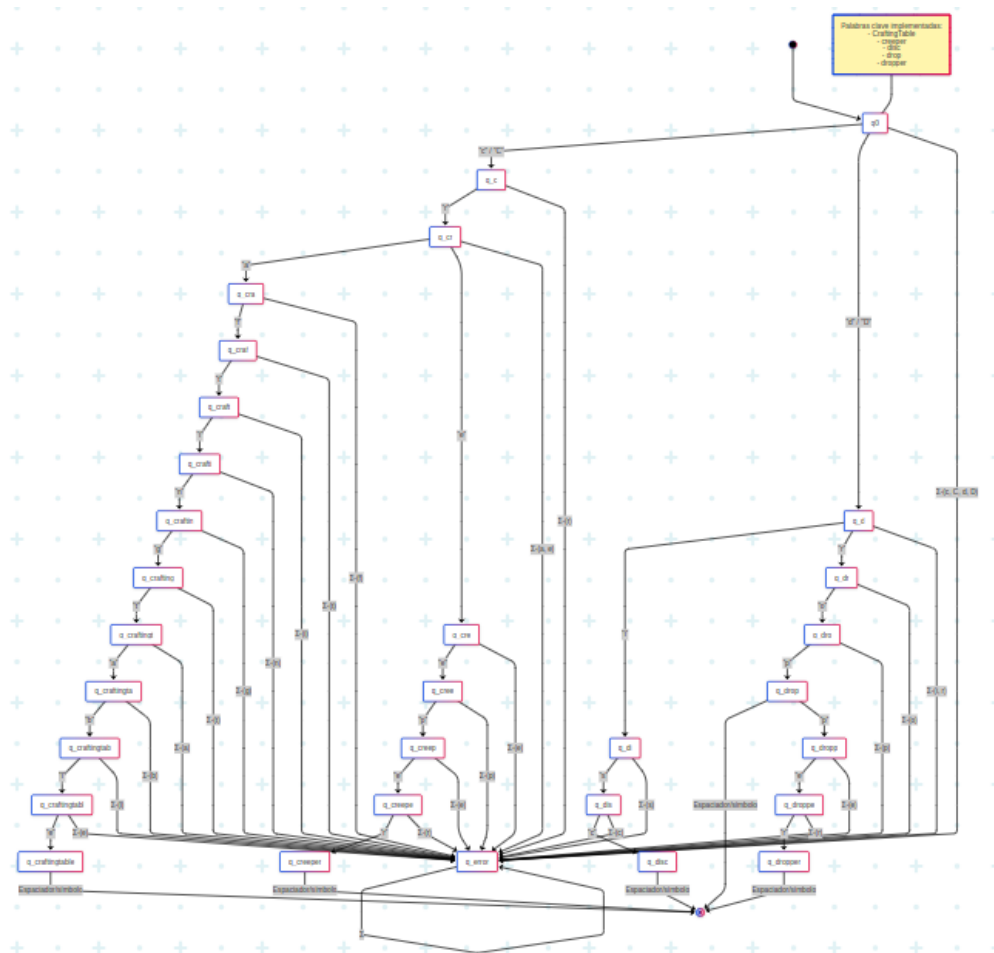


Figura 5.10: Diagrama del reconocedor de palabras reservadas de craftingTable-dropper

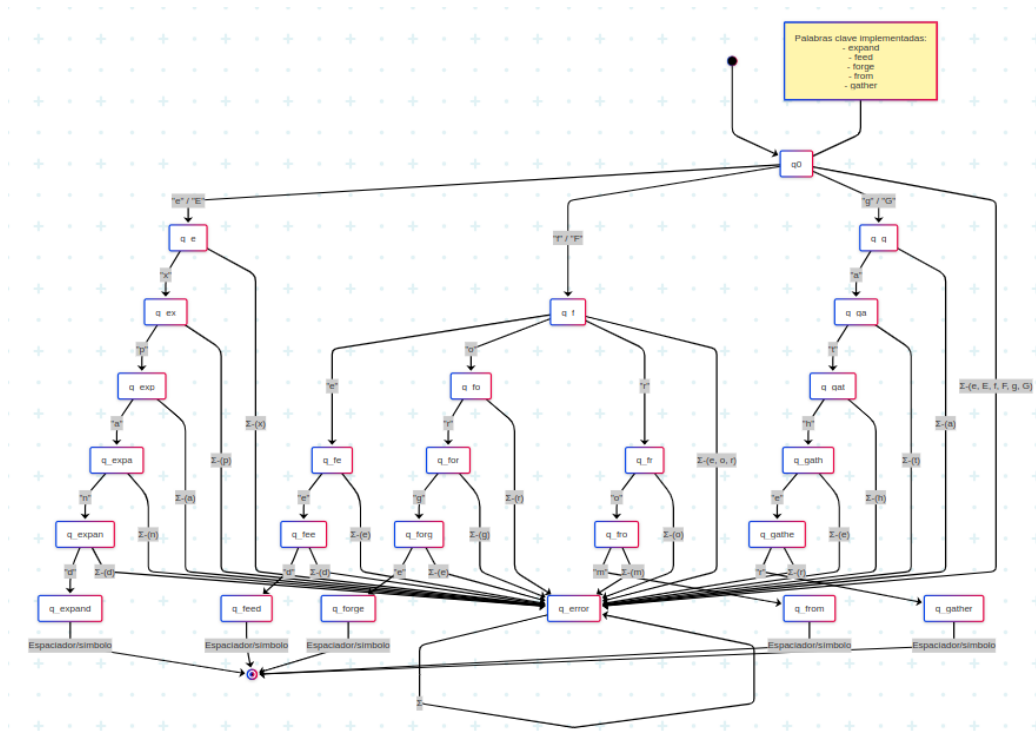


Figura 5.12: Diagrama del reconocedor de palabras reservadas de expand-gather

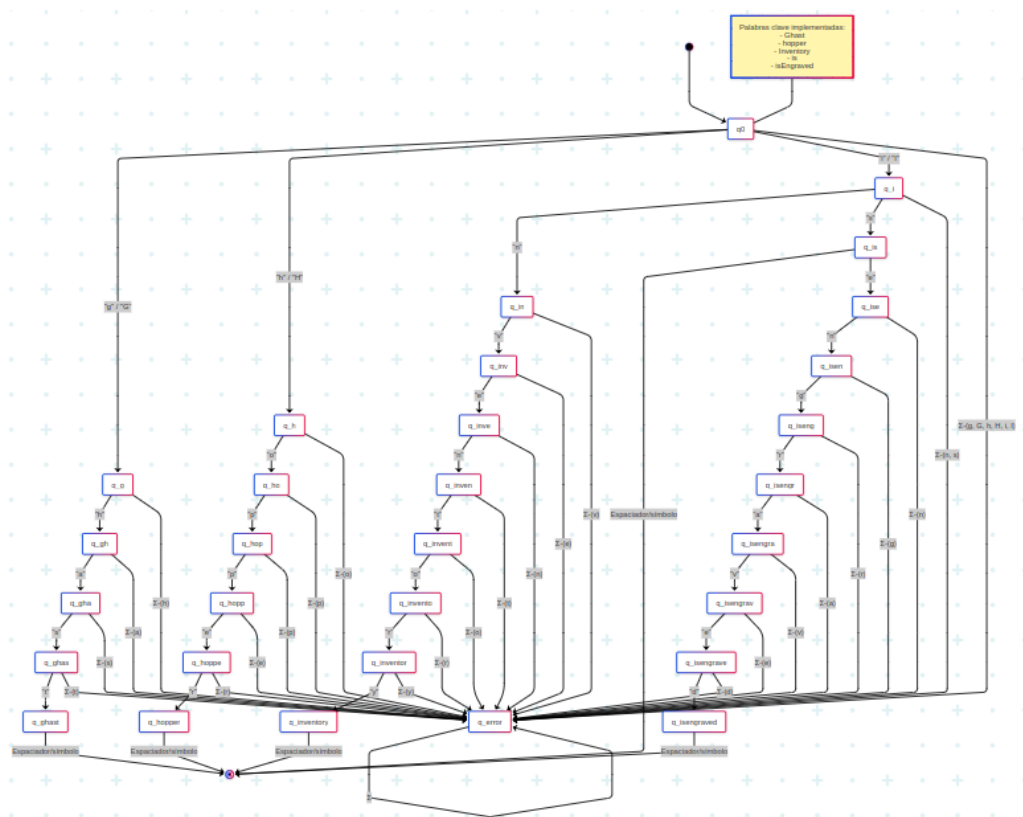


Figura 5.13: Diagrama del reconocedor de palabras reservadas de ghash-isEngraved

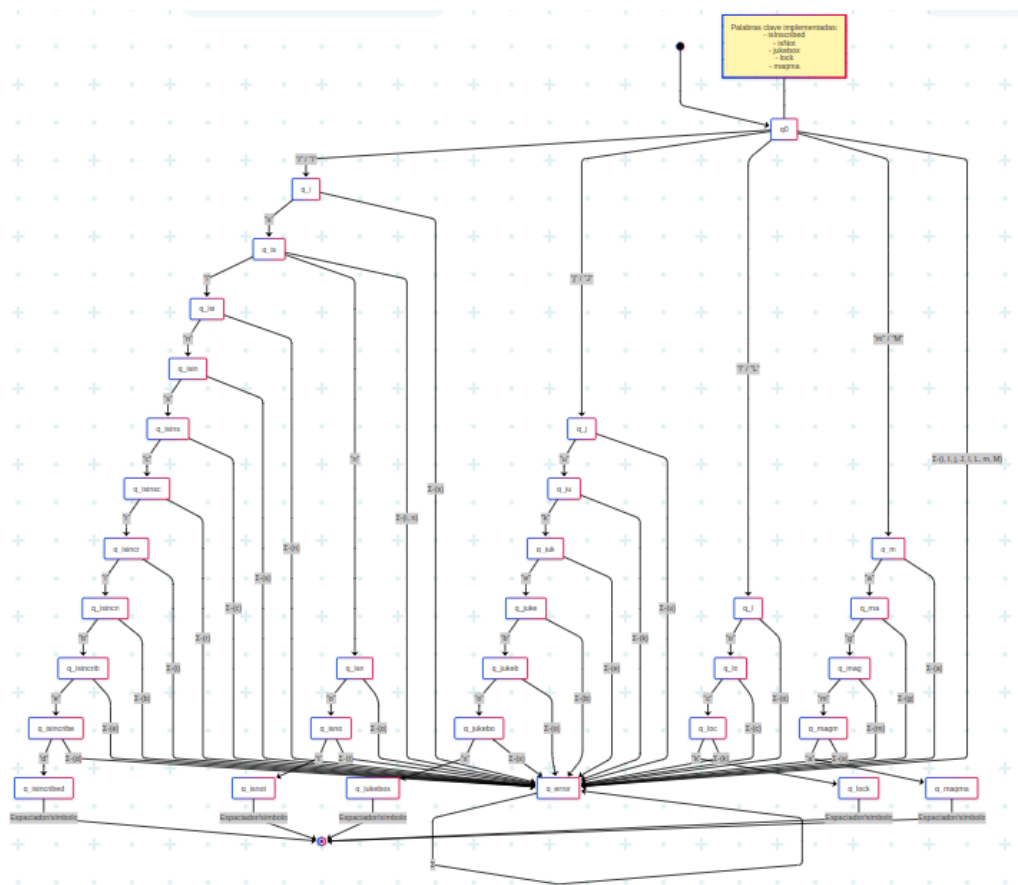


Figura 5.14: Diagrama del reconocedor de palabras reservadas de isInscribed-magma

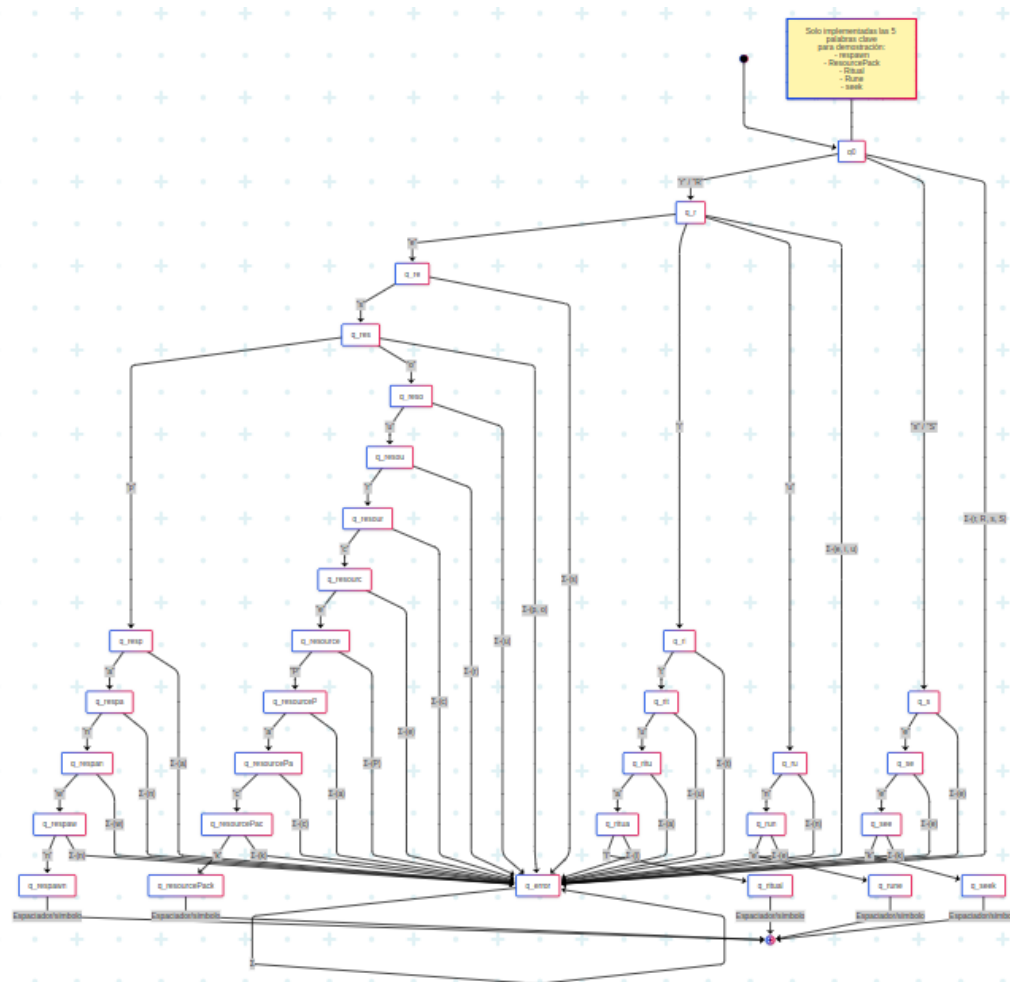


Figura 5.17: Diagrama del reconocedor de palabras reservadas de respawnseek

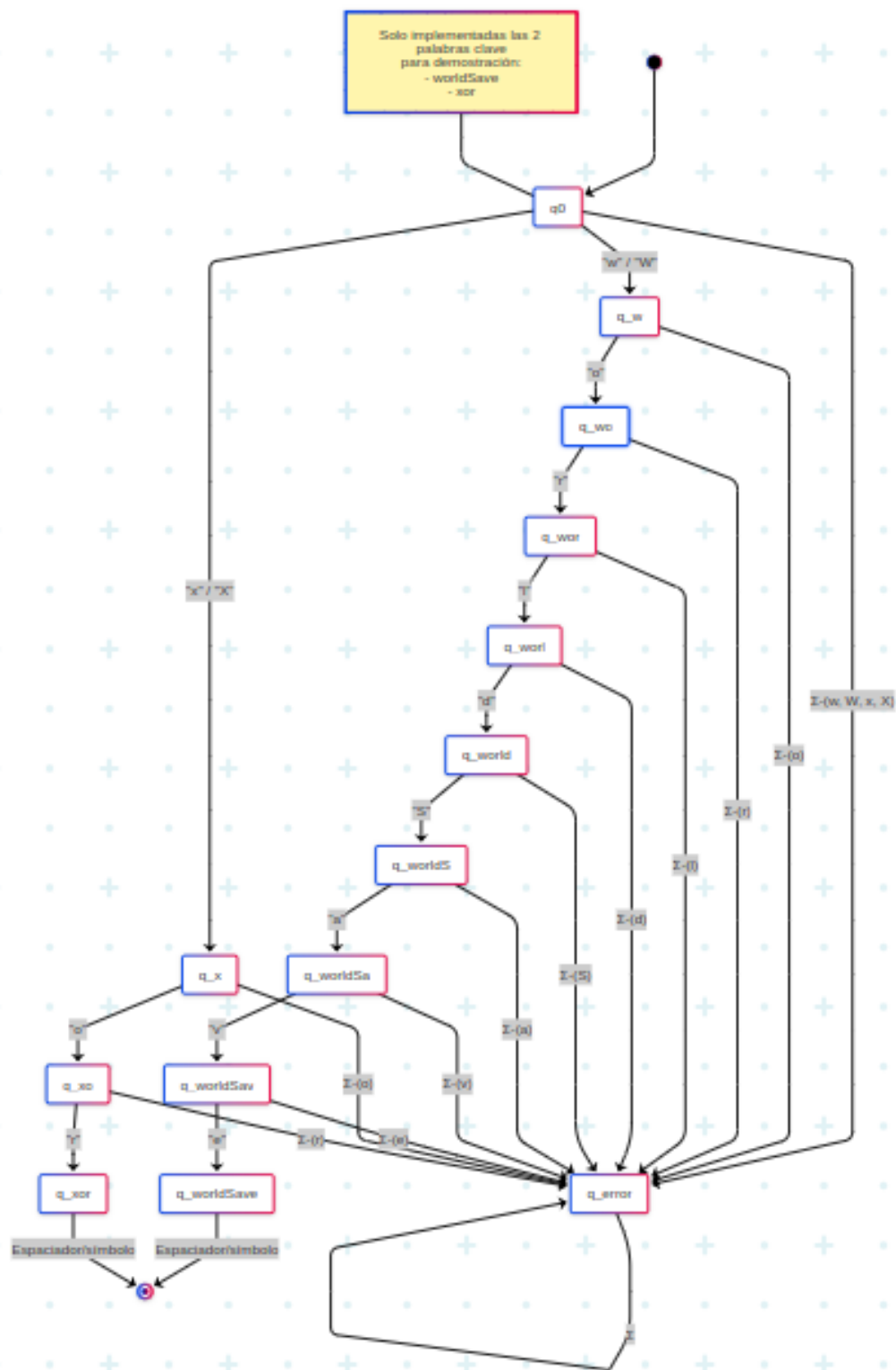


Figura 5.21: Diagrama del reconocedor de palabras reservadas de worldSave-xor

Capítulo 6

Documentacion del Scanner

6.1. Conceptos básicos del Scanner

El analizador léxico desarrollado para el lenguaje **Notch Engine** tiene como objetivo principal transformar el texto fuente de un programa en una secuencia de *tokens*, los cuales son enviados posteriormente al analizador sintáctico.

Este scanner fue implementado de manera modular, organizando los componentes principales en diferentes archivos y clases.

6.1.1. Estructura General

El scanner se compone de:

- **Core del Scanner:** (`core.py`) Contiene la clase **Scanner** que administra el proceso de lectura de caracteres, el control de estados, y la coordinación con los autómatas especializados.
- **Definición de Tokens:** (`tokens.py`) Define la clase **Token** y los tipos de tokens que el lenguaje reconoce, incluyendo palabras reservadas, literales, operadores y delimitadores.
- **Manejo de Errores:** (`error_handling.py`) Implementa un sistema robusto de captura y recuperación de errores léxicos, para asegurar que el análisis no se detenga ante errores detectados.
- **Autómatas Especializados:** Cada tipo de elemento del lenguaje (identificadores, números, cadenas, operadores y comentarios) tiene su propio autómata para reconocer patrones válidos en el flujo de entrada.

- **Herramientas de Conteo de Comentarios:** (`cantidadComentarios.py`) Funciones auxiliares para contabilizar comentarios de línea y bloque en los archivos de entrada.
- **Generador de Resultados:** (`generarMuroLadrillos.py`) Se encarga de generar un archivo HTML visualizando los tokens y las estadísticas del análisis.

6.1.2. Funcionamiento del Scanner

El scanner funciona siguiendo los siguientes pasos principales:

1. Inicialización:

- Se abre el archivo fuente.
- Se carga su contenido en memoria.
- Se preparan los autómatas para distintos tipos de tokens.
- Se inicializa el primer token para comenzar el proceso.

2. Tokenización:

- El método `deme_token()` es llamado repetidamente.
- Se prueba el carácter actual con cada autómata hasta encontrar una coincidencia.
- Si un autómata reconoce un lexema válido, se crea un nuevo **Token** con la información correspondiente.
- En caso de encontrar un carácter inválido, se registra un error de tipo léxico.

3. Manejo de Secuencias Especiales:

- El scanner realiza validaciones adicionales para detectar errores de sintaxis tempranos como falta de punto y coma al final de instrucciones o mal uso de llaves.
- Se implementa verificación automática de bloques y control de contexto (por ejemplo, bloques delimitados por `PolloCrudo` y `PolloAsado`).

4. Finalización:

- Antes de cerrar el archivo, se verifica que todas las secuencias de tokens hayan sido completadas correctamente.
- Se realiza el cierre seguro del archivo fuente.

6.1.3. Recuperación de Errores

El scanner implementa un mecanismo de recuperación de errores léxicos basado en la captura inmediata y la continuación del proceso de tokenización, evitando el pánico o la detención total del análisis.

Al detectar un error:

- Se registra el error mediante el `ErrorHandler`.
- Se genera un token de tipo `ERROR` que encapsula la información del problema.
- El scanner avanza al siguiente carácter para intentar recuperar el flujo de tokens válidos.

Esto permite detectar múltiples errores en una sola ejecución, mejorando la robustez y utilidad del analizador.

6.1.4. Reconocimiento de Comentarios

Se manejan dos tipos de comentarios de acuerdo al lenguaje Notch Engine:

- **Comentario de línea (\$\$):** Se reconocen hasta el fin de la línea actual.
- **Comentario de bloque (\$* *\$):** Se reconocen múltiples líneas, terminando cuando se encuentra el delimitador de cierre *\$.

Los comentarios son ignorados durante el análisis léxico, pero su conteo es reportado en las estadísticas finales.

6.1.5. Generación de Resultados

Después de finalizar el análisis:

- Se genera un archivo HTML donde cada token válido es representado como un ladrillo de colores.
- Se incluyen estadísticas como:
 - Cantidad de palabras reservadas
 - Cantidad de identificadores
 - Cantidad de literales numéricos y de texto
 - Cantidad de operadores
 - Número de errores léxicos encontrados
 - Número de líneas, caracteres y comentarios procesados

6.1.6. Cumplimiento de las Reglas del Proyecto

Este scanner fue desarrollado cumpliendo estrictamente las especificaciones requeridas:

- No se permite el uso de llaves para secciones (`Bedrock`, `Inventory`, etc.).
- Toda instrucción simple debe terminar en punto y coma (;).
- Implementación obligatoria de recuperación de errores.
- Reconocimiento de todos los tokens definidos para el lenguaje.
- Generación automática de un reporte visual del análisis.

6.2. Core del Scanner

El archivo `core.py` implementa el núcleo del analizador léxico, mediante la clase `Scanner`. Esta clase es la responsable de coordinar el proceso de lectura de caracteres, análisis léxico, creación de tokens y detección de errores, siguiendo las reglas establecidas por el lenguaje MC.

6.2.1. Clase Scanner

La clase `Scanner` organiza su funcionamiento mediante los siguientes atributos principales:

- **nombre_archivo:** Ruta al archivo fuente que se analiza.
- **contenido:** Contenido completo del archivo leído en memoria.
- **posicion, linea, columna:** Controlan la ubicación actual en el texto para un rastreo exacto de tokens y errores.
- **automata:** Instancia única de `IntegratedAutomaton` que centraliza toda la funcionalidad de análisis léxico.
- **token_actual, token_siguiente:** Referencias al token en procesamiento y al siguiente token para implementar un mecanismo simple de *lookahead*.
- **manejador_errores:** Instancia de `ErrorHandler` para registrar errores léxicos encontrados.

6.2.2. Clase ErrorHandler

Implementa un sistema simple para el manejo y registro de errores:

- **errores:** Lista que almacena todos los errores encontrados durante el análisis.
- **registrar_error():** Método que documenta los errores con tipo, mensaje, línea y columna.
- **hay_errores():** Método que verifica la existencia de errores registrados.

6.2.3. Métodos Principales del Scanner

- **inicializar_scanner():**
 - Abre y carga el archivo de entrada.
 - Inicializa el autómata integrado.
 - Lee el primer token para iniciar el análisis.
- **deme_token():**
 - Devuelve el token actual y avanza al siguiente.
 - Implementa un mecanismo simple de avance en la secuencia de tokens.
- **tome_token():**
 - Permite obtener el token actual sin avanzar, útil para mecanismos de *lookahead* en análisis sintáctico.
- **finalizar_scanner():**
 - Cierra el archivo fuente de manera segura.

6.2.4. Métodos Auxiliares

- **_cargar_primer_token():**
 - Inicializa la secuencia de análisis cargando el primer token.
- **_siguiente_token():**
 - Método central que coordina el proceso de análisis léxico.

- Obtiene el siguiente token del contenido a partir de la posición actual.
 - Ignora espacios en blanco y utiliza el autómata integrado para el análisis.
 - Maneja errores léxicos generando tokens de tipo **ERROR**.
- `_ignorar_espacios()`:
 - Avanza la posición actual ignorando espacios en blanco.
 - Actualiza contadores de línea y columna según corresponda.

6.2.5. Funcionamiento del Análisis Léxico

El proceso para obtener cada token sigue estos pasos:

1. Se ignoran los espacios y saltos de línea.
2. Se invoca al autómata integrado con la posición actual en el texto.
3. El autómata integrado analiza el texto y determina:
 - Tipo de token reconocido
 - Lexema correspondiente
 - Valor semántico asociado (si aplica)
 - Posición final tras consumir el token
4. Si el autómata reconoce un lexema válido, se genera el **Token** correspondiente.
5. Si el autómata no puede reconocer el símbolo, se registra un error léxico.

6.2.6. Clase `IntegratedAutomaton`

Aunque no se muestra en el código proporcionado, el scanner utiliza un autómata integrado que centraliza la funcionalidad de análisis léxico:

- Combina la funcionalidad de detección de todos los tipos de tokens.
- Devuelve un resultado estructurado con información sobre el token identificado.
- Incluye datos como éxito del análisis, tipo de token, lexema, valor asociado y posición final.

6.2.7. Recuperación de Errores

En vez de detener el análisis en el primer error encontrado, el **Scanner**:

- Genera un **Token** de tipo **ERROR** encapsulando el carácter no reconocido.
- Continúa el proceso para analizar el resto del archivo, permitiendo encontrar múltiples errores en una misma ejecución.
- Registra el error mediante el **ErrorHandler** para generar reportes posteriores.

6.2.8. Importancia del Scanner

El **Scanner** constituye una de las piezas fundamentales del compilador, ya que:

- Permite identificar los componentes léxicos válidos del lenguaje.
- Detecta y reporta errores de manera temprana.
- Sirve como base para los siguientes módulos de análisis sintáctico y semántico.
- Proporciona un mecanismo simple de *lookahead* que facilita el análisis sintáctico.

6.3. Explicación del Autómata

El scanner utiliza un autómata integrado para reconocer los distintos patrones léxicos definidos por el lenguaje Notch Engine. A diferencia de la versión anterior que utilizaba múltiples autómatas separados, la nueva implementación encapsula todas las funcionalidades en una única clase **IntegratedAutomaton**.

6.3.1. Estructura Base del Autómata Integrado

El nuevo autómata se compone de:

- **Clase AutomatonResult**: Encapsula el resultado del procesamiento de cada token, incluyendo:
 - **exito**: Indica si el procesamiento fue exitoso.
 - **tipo**: El tipo de token reconocido.

- **lexema:** La cadena de caracteres que forma el token.
 - **valor:** El valor semántico extraído del lexema.
 - **final_pos, final_linea, final_columna:** Posición final después de procesar el token.
- **Clase IntegratedAutomaton:** Implementa la lógica unificada de reconocimiento para todos los tipos de tokens.

El autómata integrado implementa el método principal:

- **procesar(contenido, posicion, linea, columna):** Método que determina qué tipo de token comienza en la posición actual y delega en métodos auxiliares para su procesamiento completo.

6.3.2. Procesamiento por Tipo de Token

El autómata integrado contiene métodos especializados para cada familia de tokens:

- **_procesar_comentario():** Maneja comentarios de línea (`$$`) y de bloque (`$* ... *$`).
- **_procesar_string():** Procesa strings (encerrados en `"`) y caracteres (encerrados en `'`).
- **_procesar_numero():** Reconoce números enteros y decimales.
- **_procesar_identificador():** Procesa identificadores y palabras reservadas.
- **_procesar_operador():** Maneja operadores y símbolos especiales.

6.3.3. Procesamiento de Comentarios

El método `_procesar_comentario` maneja:

- **Comentarios de línea:** Inician con `$$` y terminan al final de la línea.
- **Comentarios de bloque:** Inician con `$*` y terminan con `*$`.

Incorpora manejo especial para:

- Conteo correcto de saltos de línea dentro de los comentarios.
- Cálculo preciso de posición final y columna después del comentario.
- Detección de comentarios de bloque no cerrados, tratándolos como error.

6.3.4. Procesamiento de Strings y Caracteres

El método `_procesar_string` maneja:

- **Strings (CADENA):** Texto encerrado entre comillas dobles (`"`).
- **Caracteres (CHARACTER):** Texto encerrado entre comillas simples (`'`).

Incluye soporte para:

- Secuencias de escape utilizando `\`.
- Seguimiento de líneas y columnas cuando hay saltos de línea dentro de strings.
- Detección de strings o caracteres no cerrados, tratándolos como error.

6.3.5. Procesamiento de Números

El método `_procesar_numero` reconoce:

- **NUMERO_ENTERO:** Secuencias de dígitos sin punto decimal.
- **NUMERO_DECIMAL:** Secuencias numéricas que contienen un punto decimal.

El valor semántico es procesado:

- Como `int` si es un entero.
- Como `float` si es un número decimal.

Implementa validaciones para asegurar que los puntos decimales estén seguidos por dígitos.

6.3.6. Procesamiento de Identificadores

El método `_procesar_identificador` reconoce:

- **IDENTIFICADOR:** Comienzan con una letra o guion bajo (`_`), seguidos de letras, dígitos o guiones bajos.
- **Palabras reservadas:** Si un identificador coincide con una palabra reservada conocida en `PALABRAS_RESERVADAS`, se clasifica con el tipo correspondiente.

El método consulta un diccionario externo de palabras reservadas importado desde el módulo de tokens.

6.3.7. Procesamiento de Operadores y Símbolos

El método `_procesar_operador` reconoce varios tipos de operadores:

- **Operadores aritméticos:** `+`, `-`, `*`, `/**`, `%`.
- **Operadores de comparación:** `>`, `<`, `>=`, `<=`, `==`, `!=`.
- **Delimitadores:** `(`, `)`, `[`, `]`.
- **Símbolos especiales:** `;`, `,`, `.`, `:`, `#`, `@`, `->`, `::`.
- **Operadores especiales:** `:+`, `:-`, `:*`, `:/`, `:%` (operaciones flotantes).
- **Operadores de incremento/decremento:** `++`, `--`.

Prioriza el reconocimiento de operadores compuestos (de dos caracteres) antes de verificar operadores simples (de un carácter).

6.3.8. Procesamiento de Delimitadores de Bloque

El autómata incluye soporte específico para los delimitadores de bloque:

- **POLLO_CRUDO:** Representa el símbolo `{` (llave de apertura).
- **POLLO_ASADO:** Representa el símbolo `}` (llave de cierre).

Estos tokens reciben un tratamiento especial en el método principal `procesar`.

6.3.9. Flujo de Procesamiento

El flujo general de procesamiento del autómata integrado es:

1. El scanner invoca el método `procesar` con la posición actual.
2. El autómata determina el tipo potencial de token basado en el primer carácter.
3. Delega el procesamiento detallado al método especializado correspondiente.
4. Retorna un objeto `AutomatonResult` con toda la información del token reconocido.

Este diseño permite un procesamiento eficiente y modular del flujo de entrada.

6.3.10. Manejo de Errores

El autómata integrado proporciona manejo de errores para:

- **Caracteres no reconocidos:** Caracterizado con el tipo `ERROR`.
- **Strings o caracteres sin cerrar:** Detectados en `_procesar_string`.
- **Comentarios de bloque no cerrados:** Identificados en `_procesar_comentario`.
- **Formatos incorrectos de números:** Controlados en `_procesar_numero`.
- **Símbolos \$ sin seguir patrón válido:** Manejados como error en `_procesar_comentario`.

La detección de errores permite al scanner continuar el procesamiento del archivo, minimizando el impacto de un token mal formado.

6.4. Explicación Tokenizador

El tokenizador es el componente del **Scanner** encargado de transformar secuencias de caracteres reconocidas por los autómatas en instancias de la clase **Token**. Cada token contiene la información necesaria para la fase sintáctica: su tipo, su lexema (texto reconocido), su posición en el archivo fuente (línea y columna), su valor semántico opcional y su categoría. El proceso de tokenización es esencial porque permite estructurar el flujo de entrada en unidades significativas que facilitan el análisis posterior. La implementación mejorada ahora incluye categorización de tokens que ayuda a organizar y documentar mejor el lenguaje.

6.4.1. Clase Token Mejorada

La nueva implementación de la clase **Token** incluye ahora los siguientes atributos:

- **type:** Tipo específico del token (por ejemplo, `WORLD_NAME`, `PLUS`, etc.)
- **lexema:** El texto reconocido del código fuente
- **linea:** Número de línea donde fue encontrado
- **columna:** Posición de columna donde inicia el token

- **valor:** Valor semántico opcional (por ejemplo, valor numérico para enteros)
- **categoría:** Categoría a la que pertenece el token, según la enumeración `TokenCategory`

6.4.2. Categorías de Tokens

Los tokens reconocidos por el scanner se clasifican en las siguientes categorías principales, definidas mediante la enumeración `TokenCategory`:

- **Estructura del Programa:** Tokens que definen la estructura general del programa (`PROGRAM_STRUCTURE`)
 - `WORLD_NAME`, `BEDROCK`, `RESOURCE_PACK`, `SPAWN_POINT`, `WORLD_SAVE`, etc.

Tipos de datos: Palabras reservadas para tipos (`DATA_TYPES`)

- `STACK`, `RUNE`, `SPIDER`, `TORCH`, `CHEST`, `BOOK`, `GHAST`, `SHELF`, `ENTITY`, `REF`

Literales booleanas: Valores booleanos (`BOOLEAN_LITERALS`)

- `ON`, `OFF`

Delimitadores de bloques: Símbolos que delimitan bloques de código (`BLOCK_DELIMITERS`)

- `POLLO_CRUDO` (`{}`) y `POLLO_ASADO` (`}`)

Control de flujo: Estructuras de control (`FLOW_CONTROL`)

- `REPEATER`, `CRAFT`, `TARGET`, `HIT`, `MISS`

Funciones y procedimientos: Estructuras para definir funciones (`FUNCTIONS`)

- `SPELL`, `RITUAL`, `RESPAWN`

Operadores lógicos: (`LOGIC_OPERATORS`)

- `AND`, `OR`, `NOT`, `XOR`

Operadores de caracteres: Para manipulación de caracteres (`CHAR_OPERATORS`)

- `IS_ENGRAVED`, `IS_INSCRIBED`, `ETCH_UP`, `ETCH_DOWN`

Operadores de strings: Para manipulación de cadenas (`STRING_OPERATORS`)

- BIND, HASH, FROM, EXCEPT, SEEK

Operadores de conjuntos: Para manipulación de conjuntos (SET_OPERATORS)

- ADD, DROP, FEED, MAP

Operadores de archivos: Para manejo de archivos (FILE_OPERATORS)

Comparación de Operadores: (COMPARISON_OPERATORS)

- IS, IS_NOT, operadores matemáticos de comparación como >, <, >=, <=

Funciones de entrada/salida: (IO_FUNCTIONS)

Otros operadores: Operadores diversos (OTHER_OPERATORS)

- BIOM, KILL, UNLOCK, LOCK, MAKE, GATHER, FORGE, EXPAND

Operadores generales: Categoría general para operadores (OPERATORS)

- Operadores aritméticos básicos: +, −, *, //, %
- Operadores flotantes: : +, : −, : *, : //, : %

Identificadores: Nombres de variables y otros identificadores (IDENTIFIERS)

Literales: Valores constantes (LITERALS)

- Números enteros (NUMERO_ENTERO)
- Números decimales (NUMERO_DECIMAL)
- Cadenas (CADENA) reconocidas entre comillas dobles
- Caracteres (CHARACTER) reconocidos entre comillas simples

Especiales: Tokens que no encajan en otras categorías (SPECIAL)

- Incluye tokens de error (ERROR)
- Comentarios (COMENTARIO)

6.4.3. Palabras Reservadas

El lenguaje MC contiene un amplio conjunto de palabras reservadas organizadas por dominio semántico, incluyendo:

- **Términos de estructura:** `worldname`, `bedrock`, `resourcepack`, `inventory`, `recipe`, etc.
- **Tipos de datos:** `stack`, `rune`, `spider`, `torch`, `chest`, etc.
- **Estructuras de control:** `repeater`, `craft`, `target`, `hit`, `miss`, etc.
- **Operadores lógicos:** `and`, `or`, `not`, `xor`
- **Manipuladores:** `bind`, `hash`, `from`, `seek`, `add`, `drop`, etc.
- **Operadores especiales:** Como los delimitadores `(POLL0CRUDO)y(`

6.4.4. Manejo de Operadores

El manejo de operadores dentro del scanner es realizado principalmente por el `OperatorAutomaton`. Este autómata está diseñado para reconocer:

- **Operadores aritméticos básicos:** `+`, `-`, `//`
- **Operadores flotantes:** `: +`, `: -`, `: :`, `: //`
- **Operadores de comparación:** `>`, `<`, `>=`, `<=`, además de `is` e `isnot`.
- **Operadores lógicos:** Reconocidos como palabras reservadas: `and`, `or`, `not`, `xor`.
- **Operadores especiales:**
 - `>>` para la coerción de tipos.
 - `bind`, `from`, `seek`, etc., para operaciones específicas de cadenas y archivos.
- **Delimitadores:** Símbolos como paréntesis, corchetes, llaves (representados como `POLL0CRUDOy`)

Estrategias de Reconocimiento de Operadores

- El autómata identifica operadores de un solo carácter inmediatamente (+, −, etc.).
- Si detecta símbolos compuestos (//, >=, <=, : +), continúa leyendo los siguientes caracteres para formar el operador completo.
- El scanner tiene tolerancia a errores en operadores incompletos, registrándolos mediante el **ErrorHandler** si un operador esperado no se completa correctamente.

Casos Especiales

- **División de enteros:** La doble barra // debe ser reconocida como un único token (`DIVISION_ENTERA`). **Operadores flotantes:** Son operadores especiales que
- **Operadores de coerción:** El operador \gg es utilizado para reinterpretar tipos y debe ser tratado como un operador único y válido.

La correcta identificación de los operadores garantiza la construcción adecuada de las expresiones y el control de flujo durante el análisis sintáctico posterior. La categorización adicional de los tokens facilita tanto el mantenimiento del compilador como la documentación del lenguaje.

6.5. Explicación de Brickwall

6.5.1. Descripción general

La función `generarLadrillos` es una herramienta de visualización que transforma los resultados de un análisis léxico en una representación gráfica HTML. Su principal propósito es generar un "muro de ladrillos" donde cada ladrillo representa un lexema analizado, facilitando así la interpretación visual de la estructura del código fuente.

6.5.2. Parámetros de entrada

La función recibe los siguientes parámetros:

- **contenido** (lista): Colección de strings con los lexemas identificados que se mostrarán como ladrillos individuales.

- **estadisticaToken** (diccionario): Contabilización de tokens agrupados por familias.
- **lineasPrograma** (entero): Número total de líneas del código analizado.
- **numeroCaracteresEntrada** (entero): Cantidad de caracteres en el archivo de entrada.
- **numeroComentariosLinea** (entero): Total de comentarios de una sola línea detectados.
- **numeroComentariosBloque** (entero): Total de comentarios multilínea detectados.
- **cantidadErrores** (entero): Número de errores léxicos encontrados durante el análisis.

6.5.3. Estructura de salida

La función genera un archivo HTML llamado `analisis_lexico.html` con tres secciones principales:

1. **Muro de ladrillos:** Representación visual de los lexemas como bloques de colores.
2. **Estadísticas de tokens:** Listado cuantitativo de los tokens por familia.
3. **Otras estadísticas:** Métricas generales del código analizado.

6.5.4. Elementos visuales

- **Ladrillos:** Cada lexema se presenta como un elemento `div` con un color de fondo distintivo.
- **Paleta de colores:** Se utiliza una selección de 15 colores distintos que se asignan cíclicamente a los lexemas.
- **Disposición adaptativa:** El diseño del muro se ajusta automáticamente mediante `flexbox` para adaptarse a diferentes tamaños de pantalla.
- **Efectos visuales:** Los ladrillos incluyen sombras suaves y bordes redondeados para mejorar la estética.

6.5.5. Estadísticas generadas

El informe HTML incluye dos secciones de estadísticas:

- **Estadísticas de tokens:** Muestra la frecuencia de cada familia de tokens (solo cuando su conteo es mayor que cero).
- **Otras estadísticas:** Presenta las métricas generales como número de líneas, caracteres y comentarios, omitiendo aquellas con valor cero.

6.5.6. Implementación

Aspectos destacados de la implementación:

- **Seguridad HTML:** Los caracteres especiales de los lexemas son escapados (&, ¡, ¿) para evitar problemas de renderizado.
- **Generación dinámica:** El código HTML se construye mediante string formatting para insertar los datos dinámicamente.
- **Filtrado inteligente:** Solo se muestran estadísticas con valores mayores que cero, optimizando así el espacio visual.
- **Estilos responsivos:** La hoja de estilos CSS integrada incluye media queries para adaptar la visualización a dispositivos móviles.

6.5.7. Uso típico

La función se utilizaría normalmente como parte de un proceso de análisis léxico:

```
# Ejemplo de uso después de realizar un análisis léxico
generarLadrillos(
    lexemas_detectados,
    estadisticas_tokens,
    total_lineas,
    total_caracteres,
    comentarios_linea,
    comentarios_bloque,
    errores_detectados
)
```

El resultado es un archivo HTML que puede abrirse en cualquier navegador moderno para visualizar los resultados del análisis realizado.

6.6. Ejecución del Scanner

6.6.1. Explicación de Scripts

Para facilitar el desarrollo del programa en diferentes sistemas operativos, se implementaron scripts automatizados que gestionan el entorno virtual y sincronizan los archivos del proyecto. Estos scripts permiten crear, sincronizar y eliminar el entorno de desarrollo con simples comandos, garantizando una experiencia consistente independientemente del sistema operativo utilizado.

Script para Linux (entorno.sh)

El script `entorno.sh` proporciona funcionalidad para gestionar el entorno virtual en sistemas basados en Linux/Unix:

- **Crear entorno:** Genera un entorno virtual Python, crea estructura de directorios y copia los archivos fuente.
- **Sincronizar:** Actualiza los archivos originales con los cambios realizados en el entorno.
- **Eliminar:** Limpia el entorno virtual por completo.

Ejemplo de uso:

```
./scripts/entorno.sh crear      # Crea y configura el entorno
./scripts/entorno.sh sync      # Sincroniza cambios al directorio original
./scripts/entorno.sh eliminar  # Elimina el entorno
```

Script para Windows (entorno.ps1)

El script `entorno.ps1` implementa las mismas funcionalidades para sistemas Windows mediante PowerShell:

- **Crear entorno:** Genera el entorno virtual, crea directorios y copia archivos fuente.
- **Sincronizar:** Actualiza los archivos originales desde el entorno de desarrollo.
- **Eliminar:** Elimina el entorno virtual completamente.

Ejemplo de uso:

```
.\scripts\entorno.ps1 -accion crear      # Crea y configura el entorno
.\scripts\entorno.ps1 -accion sync      # Sincroniza cambios
.\scripts\entorno.ps1 -accion eliminar  # Elimina el entorno
```

Beneficios clave

- **Portabilidad:** Facilita el desarrollo en equipos con diferentes sistemas operativos.
- **Aislamiento:** El entorno virtual evita conflictos con otras dependencias instaladas.
- **Sincronización:** Permite trabajar en el entorno aislado y luego sincronizar los cambios.
- **Reproducibilidad:** Garantiza que todos los desarrolladores trabajen en condiciones idénticas.

Estos scripts fueron fundamentales para mantener la consistencia del desarrollo entre distintos entornos, minimizando problemas de compatibilidad y facilitando la colaboración entre miembros del equipo que utilizan diferentes plataformas.

6.6.2. Ejecución del Scanner

A continuación, se demuestra el funcionamiento del *Scanner* mediante una serie de imágenes que ilustran cada paso del proceso. Este componente es fundamental para analizar el contenido de los archivos fuente, identificar los distintos tokens y generar tanto salidas visuales como estadísticas relacionadas con los elementos detectados.

Selección de archivo

Lo primero que se observa es la interfaz principal del sistema, en donde se despliega un menú con diferentes opciones. Esta es la entrada principal del usuario al sistema:

```
(mi_entorno) samir-cabrera@samir-cabrera-ThinkPad-E
ain.py

=====
                MC Scanner - Menú de Pruebas
=====
1. 01_Prueba_PR_Estructura.txt
2. 02_Prueba_PR_Tipos.txt
3. 03_Prueba_PR_Booleanos.txt
4. 04_Prueba_PR_Bloques.txt
5. 05_Prueba_PR_Control.txt
6. 06_Prueba_PR_Saltos.txt
7. 07_Prueba_PR_Funciones.txt
8. 08_Prueba_PR_Operadores.txt
9. 09_Prueba_Lit_Enterros.txt
10. 10_Prueba_Lit_Flotantes.txt
11. 11_Prueba_Lit_Caracteres.txt
12. 12_Prueba_Lit_Strings.txt
13. 13_Prueba_Lit_Arreglos.txt
14. 14_Prueba_Lit_Registros.txt
15. 15_Prueba_Lit_Conjuntos.txt
16. 16_Prueba_Lit_Archivos.txt
17. 17_Prueba_Op_Aritmeticos.txt
18. 18_Prueba_Op_Flotantes.txt
19. 19_Prueba_Op_Comparacion.txt
20. 20_Prueba_Op_Asignacion.txt
21. 21_Prueba_Op_Acceso.txt
22. 22_Prueba_Op_Especiales.txt
```

Figura 6.1: Selección de menú

Desde esta interfaz, el usuario puede seleccionar uno de los archivos disponibles para ser analizado. En este caso, se elige el archivo número 1, a modo de prueba.

Proceso de análisis léxico

Una vez seleccionado el archivo, se inicia de forma automática el proceso de *scanning*. Durante esta etapa, el *Scanner* lee el contenido del archivo, lo

analiza y extrae los tokens válidos conforme a la gramática definida para el lenguaje en cuestión.

Los resultados del análisis se presentan en consola o interfaz, en forma de una lista de tokens acompañados de su tipo, posición y valor:

```
Token(type=SUMA, lexema='+', linea=35, columna=29)
Token(type=IDENTIFICADOR, lexema='b', linea=35, columna=31)
Token(type=PUNTO_Y_COMA, lexema=';', linea=35, columna=32)
Token(type=RESPAWN, lexema='respawn', linea=36, columna=9)
Token(type=IDENTIFICADOR, lexema='resultado', linea=36, columna=17)
Token(type=PUNTO_Y_COMA, lexema=';', linea=36, columna=26)
Token(type=IDENTIFICADOR, lexema='PolloAsado', linea=37, columna=5)
Token(type=RITUAL, lexema='Ritual', linea=39, columna=5)
Token(type=IDENTIFICADOR, lexema='mostrarMensaje', linea=39, columna=12)
Token(type=PARENTESIS_ABRE, lexema='(', linea=39, columna=26)
Token(type=SPIDER, lexema='Spider', linea=39, columna=27)
Token(type=DOBLE_DOS_PUNTOS, lexema='::', linea=39, columna=34)
Token(type=IDENTIFICADOR, lexema='texto', linea=39, columna=37)
Token(type=PARENTESIS_CIERRA, lexema=')', linea=39, columna=42)
Token(type=IDENTIFICADOR, lexema='PolloCrudo', linea=40, columna=5)
Token(type=DROPPER_SPIDER, lexema='dropperSpider', linea=41, columna=9)
Token(type=PARENTESIS_ABRE, lexema='(', linea=41, columna=22)
Token(type=IDENTIFICADOR, lexema='texto', linea=41, columna=23)
Token(type=PARENTESIS_CIERRA, lexema=')', linea=41, columna=28)
Token(type=PUNTO_Y_COMA, lexema=';', linea=41, columna=29)
Token(type=IDENTIFICADOR, lexema='PolloAsado', linea=42, columna=5)
Token(type=SPAWN_POINT, lexema='SpawnPoint', linea=45, columna=1)
Token(type=IDENTIFICADOR, lexema='PolloCrudo', linea=46, columna=5)
Token(type=DROPPER_SPIDER, lexema='dropperSpider', linea=47, columna=9)
Token(type=PARENTESIS_ABRE, lexema='(', linea=47, columna=22)
Token(type=IDENTIFICADOR, lexema='SALUDO', linea=47, columna=23)
Token(type=PARENTESIS_CIERRA, lexema=')', linea=47, columna=29)
Token(type=PUNTO_Y_COMA, lexema=';', linea=47, columna=30)
```

Figura 6.2: Ejemplos de tokens generados durante el análisis léxico

Generación de salida HTML

Además de mostrar los resultados en pantalla, el sistema genera automáticamente un archivo con formato `.html`, el cual contiene un resumen estructurado del análisis. Este archivo es útil para realizar revisiones posteriores o integrarlo como parte de un informe más amplio.

A continuación se muestra la comprobación de la existencia del archivo generado:

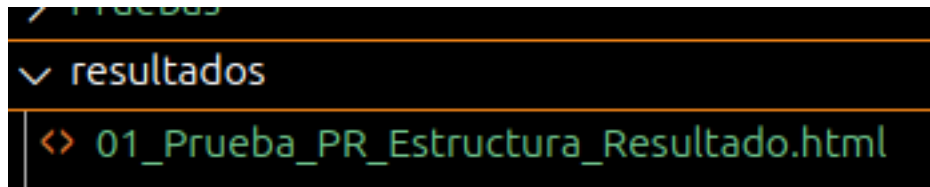


Figura 6.3: Verificación de la generación del archivo HTML

Visualización del muro de ladrillos y estadísticas

Finalmente, el sistema presenta una visualización gráfica conocida como el **muro de ladrillos** o *Brickwall*, donde cada ladrillo representa un token identificado. Esta visualización ayuda a comprender la estructura del código fuente de forma más intuitiva.



Figura 6.4: Visualización del muro de ladrillos generado por el Scanner

Asimismo, se generan estadísticas relacionadas al archivo como cantidad de palabras y algunas cantidades de tokens identificados durante el análisis. Esto puede ser de gran utilidad para detectar patrones o posibles errores en el código fuente:



Figura 6.5: Estadísticas de tokens generadas por el análisis

En conjunto, este proceso demuestra el correcto funcionamiento del *Scanner*, desde la selección del archivo hasta la visualización y almacenamiento de los resultados obtenidos.

Capítulo 7

Documentacion del Parser

7.1. Documentacion inicial

7.2. Gramática del Parser

El compilador **Notch-Engine** utiliza un parser predictivo LL(1) generado a partir de una gramática formalmente definida en formato BNF. Esta gramática fue diseñada y validada usando la herramienta **GikGram**, que permitió comprobar propiedades como factorización, ausencia de recursividad por la izquierda y unicidad de predicción.

Sección 1: Definición Inicial del Programa

Esta sección describe la estructura fundamental de un programa válido. Todo código en Notch-Engine debe comenzar con la palabra clave **WORLD_NAME**, seguida del identificador del mundo y el símbolo **:**. A continuación, se definen las secciones que componen el programa. Finalmente, se cierra con la palabra clave **WORLD_SAVE**.

Se incorporan símbolos semánticos especiales:

- **#init_tsg** – Inicializa la tabla de símbolos global.
- **#free_tsg** – Libera recursos y limpia la tabla de símbolos.

Producciones

```
<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>

```

Descripción

Cada sección define una parte específica del lenguaje:

- BEDROCK: Declaración de constantes.
- RESOURCE_PACK: Definición de tipos personalizados.
- INVENTORY: Declaración de variables globales o entidades.
- RECIPE: Prototipos de funciones o rituales.
- CRAFTING_TABLE: Implementaciones reales de funciones/procedimientos.
- SPAWN_POINT: Bloque principal de instrucciones ejecutables.

Ejemplo

```

WorldName NetherRealm:
  Bedrock
    Obsidian Stack lava = 5;
  Inventory
    Stack bucket;
  Recipe
    Spell calentar(Stack :: a) -> Stack;
  CraftingTable
    Spell calentar(Stack :: a) -> Stack {
      return a + 1;
    }

```

```
SpawnPoint
    bucket = calentar(lava);
WorldSave
```

Esta estructura garantiza que el compilador procese el código fuente de forma organizada y modular, facilitando el análisis semántico y la ejecución posterior.

Sección 2: Variables y Constantes

Esta sección abarca las producciones relacionadas con la declaración de constantes, tipos personalizados y variables. En Notch-Engine, estos elementos forman la base del entorno de ejecución del programa.

Las secciones involucradas son:

- BEDROCK – Sección de constantes (‘constantes inmutables’).
- RESOURCE_PACK – Sección de definición de tipos personalizados.
- INVENTORY – Declaraciones de variables globales o entidades.

Símbolos semánticos utilizados

- #chk_const_existence – Verifica si la constante ya existe.
- #add_const_symbol – Registra la constante en la tabla de símbolos.
- #chk_type_existence, #add_type_symbol – Validan y almacenan tipos definidos.
- #save_current_type – Guarda el tipo para asociarlo a las variables.
- #chk_var_existence, #add_var_symbol – Verifican e insertan variables.
- #mark_var_initialized – Marca la variable como inicializada.
- #default_uninitialized – Valor semántico por defecto si no hay inicialización.

Producciones

```
<bedrock_section> ::= <constant_decl> <bedrock_section>
<bedrock_section> ::=

<constant_decl> ::= OBSIDIAN <type>
                        #chk_const_existence IDENTIFICADOR <value>
                        #add_const_symbol PUNTO_Y_COMA

<value> ::= <literal>
<value> ::=

<resource_pack_section> ::= <type_decl> <resource_pack_section>
<resource_pack_section> ::=

<type_decl> ::= ANVIL #chk_type_existence #start_type_def
                IDENTIFICADOR FLECHA <type>
                #end_type_def #add_type_symbol PUNTO_Y_COMA

<inventory_section> ::= <var_decl> <inventory_section>
<inventory_section> ::=

<var_decl> ::= <type> #save_current_type <var_list> PUNTO_Y_COMA
<var_list> ::= #chk_var_existence IDENTIFICADOR <var_init>
                #add_var_symbol <more_vars>

<var_init> ::= IGUAL <expression> #mark_var_initialized
<var_init> ::= #default_uninitialized

<more_vars> ::= COMA #chk_var_existence IDENTIFICADOR <var_init>
                #add_var_symbol <more_vars>
<more_vars> ::=
```

Descripción

- Las constantes se declaran con la palabra clave OBSIDIAN, indicando tipo y valor.
- Los tipos definidos por el usuario usan la sintaxis ANVIL NombreTipo ->TipoBase; y se almacenan con su jerarquía.
- Las variables se declaran indicando su tipo y una lista de identificados.

res, con opción de inicialización.

Ejemplo

Bedrock

```
Obsidian Stack lava = 5;  
Obsidian Rune fire;
```

Resource_Pack

```
Anvil StackPower -> Stack;
```

Inventory

```
Stack bucket;  
Stack x = lava, y;
```

Esta sección establece las bases semánticas del entorno, definiendo datos inmutables, estructuras personalizadas y variables disponibles durante la ejecución.

Sección 3: Funciones y Prototipos

Esta sección define tanto los prototipos (declaraciones) como las implementaciones de funciones y procedimientos. En Notch-Engine, los métodos con valor de retorno se definen con la palabra clave **SPELL**, mientras que los procedimientos (sin retorno) se definen con **RITUAL**.

Las funciones se declaran en la sección **RECIPE** y se implementan en la sección **CRAFTING_TABLE**. Esto permite una separación clara entre la interfaz y la implementación, similar a los encabezados y cuerpos en lenguajes tradicionales.

Símbolos semánticos utilizados

- **#chk_func_start** – Marca el inicio de una implementación de función.
- **#save_func_name** – Guarda el identificador de la función para validación de retorno.
- **#set_in_function**, **#unset_in_function** – Activan contexto de función.
- **#chk_func_return** – Verifica que la función tenga retorno apropiado.

- `#create_tsl`, `#free_tsl` – Manejan la tabla de símbolos local para funciones.

Producciones

```

<recipe_section> ::= <prototype> <recipe_section>
<recipe_section> ::=

<prototype> ::= SPELL IDENTIFICADOR PARENTESIS_ABRE <params>
                PARENTESIS_CIERRA FLECHA <type> PUNTO_Y_COMA

<prototype> ::= RITUAL <proc_prototype_tail>

<proc_prototype_tail> ::= IDENTIFICADOR PARENTESIS_ABRE <params>
                        PARENTESIS_CIERRA <proc_ending>

<proc_ending> ::= #create_tsl <block> #free_tsl
<proc_ending> ::= PUNTO_Y_COMA

<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 #create_tsl #set_in_function
                    #save_func_name IDENTIFICADOR PARENTESIS_ABRE <params>
                    PARENTESIS_CIERRA FLECHA <type> <block>
                    #chk_func_return #unset_in_function #free_tsl

<proc_impl_tail> ::= IDENTIFICADOR PARENTESIS_ABRE <params>
                    PARENTESIS_CIERRA <block>

```

Descripción

- Las funciones usan la palabra clave `SPELL` e indican tipo de retorno.
- Los procedimientos usan la palabra clave `RITUAL` y no devuelven valores.
- Se admiten múltiples parámetros agrupados por tipo usando el operador `::`.

- Las funciones pueden declararse (prototipo) sin implementación inmediata.

Ejemplo

Recipe

```
Spell calentar(Stack :: a) -> Stack;
Ritual mostrarMensaje(Stack :: mensaje);
```

CraftingTable

```
Spell calentar(Stack :: a) -> Stack {
    return a + 1;
}

Ritual mostrarMensaje(Stack :: mensaje) {
    MAKE("output.txt", 'a');
    FEED(mensaje);
}
```

Gracias al uso de símbolos semánticos, se garantiza que cada función defina correctamente sus parámetros, maneje su propia tabla de símbolos local y verifique el retorno de acuerdo con su tipo declarado.

Sección 4: Sentencias y Control de Flujo

La sección de sentencias describe las instrucciones que pueden ejecutarse dentro de un bloque de código, como asignaciones, llamadas a funciones, instrucciones de control (condicionales y bucles), estructuras propias del lenguaje y bloques anidados. Todas estas sentencias son parte esencial del cuerpo ejecutable definido en la sección `SPAWN_POINT` o dentro de funciones/procedimientos.

Símbolos semánticos utilizados

- `#chk_in_loop` – Valida si una instrucción como `CREEPER` está dentro de un bucle.
- `#chk_return_context` – Valida que `RETURN` se use dentro de una función o procedimiento.
- `#mark_has_return` – Marca que una función contiene al menos un retorno.

- #chk_dead_code – Detecta código inalcanzable dentro de bloques.
- #chk_file_literal – Verifica la validez de literales de archivo en sentencias como MAKE.

Producciones Generales de Sentencias

```

<spawn_point_section> ::= <statement> <spawn_point_section>
<spawn_point_section> ::=

<statement> ::= RAGEQUIT PUNTO_Y_COMA
<statement> ::= RESPAWN <expression> PUNTO_Y_COMA
<statement> ::= MAKE PARENTESIS_ABRE <file_literal>
                    #chk_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> ::= #chk_in_loop CREEPER PUNTO_Y_COMA
<statement> ::= #chk_in_loop ENDER_PEARL PUNTO_Y_COMA
<statement> ::= #chk_return_context RETURN <return_expr>
                    #mark_has_return PUNTO_Y_COMA
<statement> ::= <block>

<block> ::= POLLO_CRUDO <statements> POLLO_ASADO
<statements> ::= <statement> #chk_dead_code <statements>
<statements> ::=

```

Producciones de Control de Flujo

```

<if_stmt> ::= TARGET <expression> #chk_bool_expr
                    CRAFT HIT <statement> <else_part>

<else_part> ::= MISS <statement> #chk_no_nested_else

```



```

<else_part> ::= #no_else

<while_stmt> ::= #enter_loop REPEATER <expression>
                #chk_bool_expr CRAFT <statement> #exit_loop

<repeat_stmt> ::= #enter_loop SPAWNER <statement>
                EXHAUSTED <expression>
                #chk_bool_expr #exit_loop PUNTO_Y_COMA

<for_stmt> ::= #enter_loop WALK #chk_for_var IDENTIFICADOR
                SET <expression> #chk_for_expr
                TO <expression> #chk_for_expr
                <step_part> CRAFT <statement> #exit_loop

<step_part> ::= STEP <expression> #chk_step_expr
<step_part> ::= #default_step_one

<switch_stmt> ::= #sw1 JUKEBOX <expression> #save_switch_type
                CRAFT <case_list> <default_case> #sw3

<case_list> ::= DISC <literal> #chk_case_type DOS_PUNTOS
                <statement> <case_list>
<case_list> ::=

<default_case> ::= #sw2 SILENCE DOS_PUNTOS <statement>

<with_stmt> ::= WITHER #chk_with_var IDENTIFICADOR
                #enter_with_scope CRAFT <statement>
                #exit_with_scope

```

Descripción

- TARGET...MISS: Condicional if...else.
- REPEATER: Bucle while.
- SPAWNER...EXHAUSTED: Bucle tipo do-while.
- WALK...STEP: Bucle tipo for.
- JUKEBOX...DISC/SILENCE: Estructura switch-case-default.
- WITHER: Control de contexto como en with...do.

- CREEPER, ENDER_PEARL: Actúan como `break` y `continue`.
- RETURN: Retorno dentro de funciones o procedimientos.
- POLLO_CRUDO y POLLO_ASADO: Delimitadores de bloques de código.

Ejemplo

```
SpawnPoint
  lava = 10;
  WALK i SET 0 TO 10 STEP 1 CRAFT {
    if TARGET i IS NOT lava CRAFT HIT {
      MAKE("error.txt", 'w');
    } MISS {
      RETURN lava;
    }
  }
```

Esta sección representa el núcleo de ejecución del lenguaje, permitiendo construir rutinas lógicas complejas con control de flujo estructurado, validado mediante acciones semánticas.

Sección 5: Asignaciones y Expresiones

En Notch-Engine, las asignaciones permiten modificar el valor asociado a una variable o estructura de datos. La parte izquierda representa un acceso a memoria válido y modificable, y la parte derecha una expresión evaluable. Las expresiones, por su parte, permiten realizar operaciones aritméticas, flotantes, lógicas y relacionales, con soporte para coerción de tipos y operadores especiales.

Símbolos semánticos utilizados

- `#chk_lvalue_modifiable` – Valida que el acceso pueda recibir asignación.
- `#push_lvalue_type` – Guarda el tipo del valor a sobrescribir.
- `#chk_assignment_types` – Verifica compatibilidad entre tipos en asignaciones.
- `#chk_float_assign_op`, `#chk_int_assign_op` – Validan operadores según tipo.

- `#push_type`, `#pop_two_push_result` – Gestionan tipos intermedios durante expresiones.
- `#chk_div_zero` – Previene divisiones por cero.
- `#apply_coercion` – Realiza coerción explícita de tipos.

Producciones de Asignación

```

<assignment> ::= <access_path>
                #chk_lvalue_modifiable
                #push_lvalue_type
                <assign_op>
                <expression>
                #chk_assignment_types

<assign_op> ::= IGUAL
<assign_op> ::= SUMA_FLOTANTE_IGUAL          #chk_float_assign_op
<assign_op> ::= RESTA_FLOTANTE_IGUAL          #chk_float_assign_op
<assign_op> ::= MULTIPLICACION_FLOTANTE_IGUAL #chk_float_assign_op
<assign_op> ::= DIVISION_FLOTANTE_IGUAL       #chk_float_assign_op
<assign_op> ::= MODULO_FLOTANTE_IGUAL         #chk_float_assign_op
<assign_op> ::= SUMA_IGUAL                    #chk_int_assign_op
<assign_op> ::= RESTA_IGUAL                    #chk_int_assign_op
<assign_op> ::= MULTIPLICACION_IGUAL          #chk_int_assign_op
<assign_op> ::= DIVISION_IGUAL                #chk_int_assign_op
<assign_op> ::= MODULO_IGUAL                  #chk_int_assign_op

```

Producciones de Expresiones

```

<expression> ::= <special_expr>

<special_expr> ::= ETCH_UP(PARENTESIS_ABRE <expression> #chk_etch_up_args
                          PARENTESIS_CIERRA)
<special_expr> ::= ETCH_DOWN(...) % similar para otras funciones especiales
...
<special_expr> ::= <numeric_expr>

<numeric_expr> ::= <common_expr>

<common_expr> ::= <logical_expr>
<common_expr> ::= <arithmetic_expr>

```

<common_expr> ::= <float_expr>

Expresiones Lógicas y Relacionales

<logical_expr> ::= <relational_expr> #push_bool_type <logical_tail>

<logical_tail> ::= XOR <relational_expr> #chk_bool_ops <logical_tail>

<logical_tail> ::= AND ...

<logical_tail> ::= OR ...

<logical_tail> ::=

<relational_expr> ::= <arithmetic_expr> <relational_tail>

<relational_tail> ::= <rel_op> <arithmetic_expr> <relational_tail>

<relational_tail> ::=

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

Expresiones Aritméticas y Flotantes

<arithmetic_expr> ::= <term> #push_type <arithmetic_tail>

<arithmetic_tail> ::= SUMA <term> #push_type #pop_two_push_result

<arithmetic_tail>

<arithmetic_tail> ::= RESTA <term> ...

<arithmetic_tail> ::=

<term> ::= <factor> <term_tail>

<term_tail> ::= MULTIPLICACION <factor> #push_type #pop_two_push_result

<term_tail>

<term_tail> ::= DIVISION <factor> #chk_div_zero ...

<term_tail> ::=

<float_expr> ::= <float_term> <float_tail>

<float_tail> ::= SUMA_FLOTANTE <float_term> #push_float_type #chk_float_ops

<float_tail>

<float_tail> ::= RESTA_FLOTANTE ...

<float_tail> ::=

<float_term> ::= <float_factor> <float_term_tail>

<float_term_tail> ::= MULTIPLICACION_FLOTANTE <float_factor> #chk_float_ops ...

<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>

```

Unarios, Coerción y Primarios

```

<factor> ::= <unary_op> <factor> #chk_unary_types
<factor> ::= <primary> <coercion_tail>

```

```

<coercion_tail> ::= COERCION <type> #apply_coercion <coercion_tail>
<coercion_tail> ::=

```

```

<primary> ::= PARENTESIS_ABRE <expression> PARENTESIS_CIERRA
<primary> ::= <literal>
<primary> ::= <id_expr>

```

```

<unary_op> ::= SUMA #push_unary_plus
<unary_op> ::= RESTA #push_unary_minus
<unary_op> ::= NOT #push_unary_not

```

Descripción

- Se admite una amplia gama de operadores de asignación para enteros y flotantes.
- Las expresiones soportan operadores booleanos, relacionales, aritméticos y funciones especiales.
- Se incluye verificación de tipos y coerción explícita con ::.
- Se detectan errores semánticos comunes como división por cero o asignaciones no compatibles.

Ejemplo

```

SpawnPoint
  a = 10;
  b = a + 2 * 5;
  f :+ 3.14;

```

```

g ::= f + 1.5;
x ::= (f + 2.5) :: Stack;

```

Gracias al soporte de expresiones enriquecidas, el lenguaje permite implementar lógica compleja con control de tipos riguroso, favoreciendo tanto seguridad como expresividad.

Sección 6: Acceso, Tipos, Llamadas y Literales

Esta sección describe cómo el lenguaje Notch-Engine permite el acceso a identificadores complejos (arreglos, registros, campos), la definición de parámetros formales para funciones, las llamadas a funciones (tanto genéricas como especiales), los tipos de datos válidos, y la representación de valores constantes o literales.

Acceso a estructuras y parámetros

El acceso a estructuras se realiza a través de rutas jerárquicas (*'access_{path}'*) *que permiten entrar a tanto en prototipos como en implementaciones.*

Símbolos semánticos utilizados:

- `#chk_id_exists` – Verifica existencia del identificador.
- `#chk_array_access` – Verifica validez de índices en acceso a arreglos.
- `#chk_record_access` – Verifica campos válidos de registros.
- `#save_param_type`, `#add_param_symbol`, `#process_param_group` – Controlan la entrada de parámetros en funciones.

```

<access_path> ::= #chk_id_exists #chk_var_initialized IDENTIFICADOR
<access_tail>
<access_tail> ::= ARROBA IDENTIFICADOR <access_tail>
<access_tail> ::= CORCHETE_ABRE <expression> #chk_array_access
CORCHETE_CIERRA <access_tail>
<access_tail> ::= PUNTO #chk_record_access IDENTIFICADOR <access_tail>
<access_tail> ::=

<params> ::= <param_group> #process_param_group <more_params>
<params> ::= #no_params

```

```

<more_params> ::= COMA <param_group> <more_params>
<more_params> ::=

<param_group> ::= <type> #save_param_type DOS_PUNTOS DOS_PUNTOS
<param_list>
<param_list> ::= #add_param_symbol IDENTIFICADOR <more_param_ids>
<more_param_ids> ::= COMA #add_param_symbol IDENTIFICADOR <more_param_ids>
<more_param_ids> ::=

```

Llamadas a funciones

Notch-Engine soporta tanto llamadas tradicionales como llamadas a funciones especiales con nombres inspirados en el entorno de Minecraft. Todas las funciones aceptan argumentos entre paréntesis, separados por comas.

Símbolos semánticos utilizados:

- #chk_func_exists, #chk_recursion, #chk_func_params – Validación de funciones y parámetros.
- #count_arg, #check_arg_count – Control del conteo de argumentos.

```

<func_call> ::= #chk_func_exists #chk_recursion IDENTIFICADOR
                PARENTESIS_ABRE <args> PARENTESIS_CIERRA
                #chk_func_params

```

```

<func_call> ::= DROPPER_STACK(PARENTESIS_ABRE <expression>
#chk_dropper_stack_args PARENTESIS_CIERRA)
<func_call> ::= ... % Todas las variantes de DROPPER y HOPPER

```

```

<args> ::= <expression> #count_arg <more_args>
<args> ::= #no_args

```

```

<more_args> ::= COMA <expression> #count_arg <more_args>
<more_args> ::= #check_arg_count

```

Tipos del lenguaje

El lenguaje incluye un sistema de tipos fuertemente inspirado en estructuras Minecraft, permitiendo referencias, arreglos y anidamiento.

Símbolos semánticos:

- #process_ref_type – Marca un tipo como referencia.

```

<type> ::= REF <type> #process_ref_type
<type> ::= STACK | RUNE | SPIDER | TORCH | CHEST | BOOK | GHAST
<type> ::= SHELF CORCHETE_ABRE <expression> CORCHETE_CIERRA <type>
<type> ::= ENTITY

```

Literales y estructuras de datos

Los valores literales pueden ser primitivos (números, cadenas, booleanos), o estructurados (arreglos, registros, conjuntos, archivos). Estas formas permiten construir estructuras complejas directamente en código fuente.

Símbolos semánticos relevantes:

- #push_int_type, #push_float_type, #push_string_type, #push_bool_type, etc.
- #start_array_literal, #start_record_literal, #end_record_literal, etc.

```

<literal> ::= CORCHETE_ABRE #start_array_literal <array_elements>
              #end_array_literal CORCHETE_CIERRA

```

```

<literal> ::= LLAVE_ABRE <literal_contenido>

```

```

<literal_contenido> ::= #start_record_literal <record_fields>
                        #end_record_literal LLAVE_CIERRA

```

```

<literal_contenido> ::= <literal_llave>

```

```

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

```

```

<literal> ::= NUMERO_ENTERO      #push_int_type
<literal> ::= NUMERO_DECIMAL    #push_float_type
<literal> ::= CADENA            #push_string_type
<literal> ::= CARACTER          #push_char_type
<literal> ::= ON                 #push_bool_type
<literal> ::= OFF                #push_bool_type

```

Elementos de estructuras

```

<set_elements> ::= <expression> <more_set_elements>
<more_set_elements> ::= COMA <expression> <more_set_elements>
<more_set_elements> ::=

```



```

<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_fields>
<more_record_fields> ::= COMA IDENTIFICADOR DOS_PUNTOS <expression>
<more_record_fields>
<more_record_fields> ::=

<file_literal> ::= CADENA COMA CARACTER

```

Ejemplo completo

Resource_Pack

```
Anvil PociónFuerte -> Stack;
```

Inventory

```
Stack x, y;
Entity Boss {
    vida;
    daño;
}
```

Recipe

```
Spell sumar(Stack :: a, b) -> Stack;
```

CraftingTable

```
Spell sumar(Stack :: a, b) -> Stack {
    return a + b;
}
```

SpawnPoint

```
x = [1, 2, 3];
y = ::{x: 10, y: 20};
Boss jefe = {
    vida: 100,
    daño: 45
};
```

Gracias a estas producciones, el lenguaje permite construir estructuras de datos ricas, funciones reutilizables y validación de tipos estricta en tiempo de análisis semántico.

Sección 7: Retornos y Resultados del Análisis LL(1)

Las expresiones de retorno (*'return_eexpr'*) *forman parte crítica de cualquier función o procedimiento*

Símbolos semánticos utilizados

- `#chk_return_in_function` – Verifica que el retorno ocurre dentro de una función válida.
- `#chk_return_in_procedure` – Verifica si un retorno está mal ubicado en un RITUAL.
- `#mark_has_return` – Marca que la función tiene al menos un retorno ejecutable.

Producciones

```
<statement> ::= #chk_return_context RETURN <return_expr>
               #mark_has_return PUNTO_Y_COMA
```

```
<return_expr> ::= #chk_return_in_function <expression>
<return_expr> ::= #chk_return_in_procedure
```

Descripción

- Si el `return` aparece dentro de una función, debe ir seguido de una expresión evaluable y compatible con el tipo de retorno declarado.
- Si el `return` aparece en un procedimiento (RITUAL), se reporta un error semántico mediante `#chk_return_in_procedure`.
- El símbolo `#mark_has_return` permite alertar si una función nunca retorna un valor.

Ejemplo

```
Spell calcular(Stack :: a) -> Stack {
  if TARGET a IS 0 CRAFT HIT {
    return 1;
  }
```

```

    } MISS {
        return a * calcular(a - 1);
    }
}

```

Sección 8: Resultados de Validación con GikGram

Durante el desarrollo de la gramática del compilador Notch-Engine, se utilizó la herramienta **GikGram** para validar la propiedad LL(1), identificar ambigüedades y generar la tabla de análisis predictivo.

```

C:\> 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 7.1: Reporte de validación generado por GikGram

Principales errores corregidos con GikGram

- **Recursividad por la izquierda:** Se refactorizaron reglas que contenían llamadas a sí mismas como primer elemento.
- **Conflictos de predicción (doble entrada por celda):** Se dividieron reglas y se forzó factorización manual sin usar ‘—’, manteniendo compatibilidad con GikGram.
- **No terminales no alcanzables:** Se eliminaron reglas huérfanas no conectadas al símbolo inicial.
- **Epsilon mal definido:** Se ajustaron reglas para admitir producciones vacías donde era sintácticamente válido.
- **Símbolos terminales mal nombrados o repetidos:** Se normalizó el vocabulario terminal en `TokenMap.py`.

Resumen del análisis LL(1)

- Total de reglas analizadas: **+250**
- No terminales definidos: **78**
- Terminales definidos: **133**
- Símbolos semánticos utilizados: **¿60**, incluyendo chequeo de tipos, contexto, estructuras y coerción.
- Estado final: **Aceptada como LL(1)** y funcional en el parser predictivo.

Conclusión

La gramática del compilador Notch-Engine fue cuidadosamente diseñada para reflejar la estructura lógica y semántica del lenguaje, asegurando un análisis sintáctico robusto y predecible. Gracias a la integración de símbolos semánticos precisos, se logra una verificación en tiempo de compilación que previene errores comunes y favorece un desarrollo seguro.

La herramienta **GikGram** fue fundamental en la validación, corrección e integración del parser. Su uso sistemático permitió asegurar que todas las producciones fueran compatibles con un análisis LL(1), sin recursividad, ambigüedad ni conflictos de predicción.

ectionDocumentación del código

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

7.2.1. 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.

7.2.2. 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.

7.2.3. 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:

7.2.4. 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.

7.2.5. 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.

7.2.6. 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.

7.2.7. 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.

7.2.8. 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.

7.2.9. Tabla Follows

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

7.2.10. 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.

7.3. 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

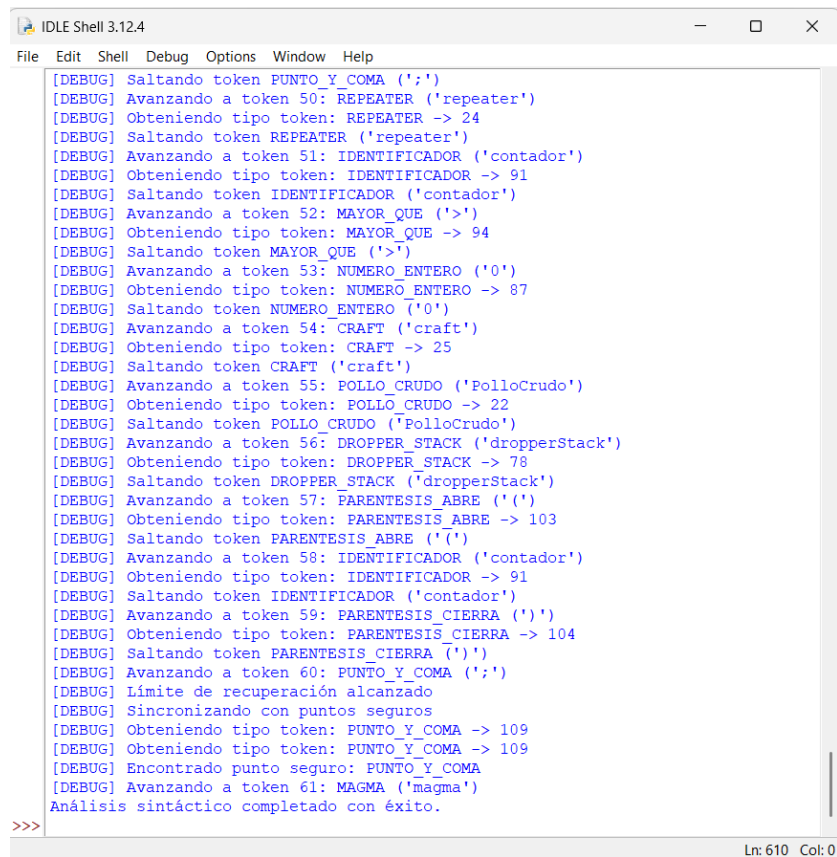
- **07_Prueba_PR_Funciones.txt**: Evalúa el manejo de declaraciones e invocaciones de funciones (**Spell**) y procedimientos (**Ritual**), incluyendo retorno con **respawn**, parámetros múltiples, llamados anidados y estructuras de control dentro del cuerpo de las funciones. Esta prueba pasó sin errores, 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 **wither**. El archivo fue aceptado correctamente, confirmando el soporte completo de instrucciones de control en el parser.

Capturas de pantalla:

```
[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')
Análisis sintáctico completado con éxito.
>>>
```

Ln: 914 Col: 0

Figura 7.2: 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

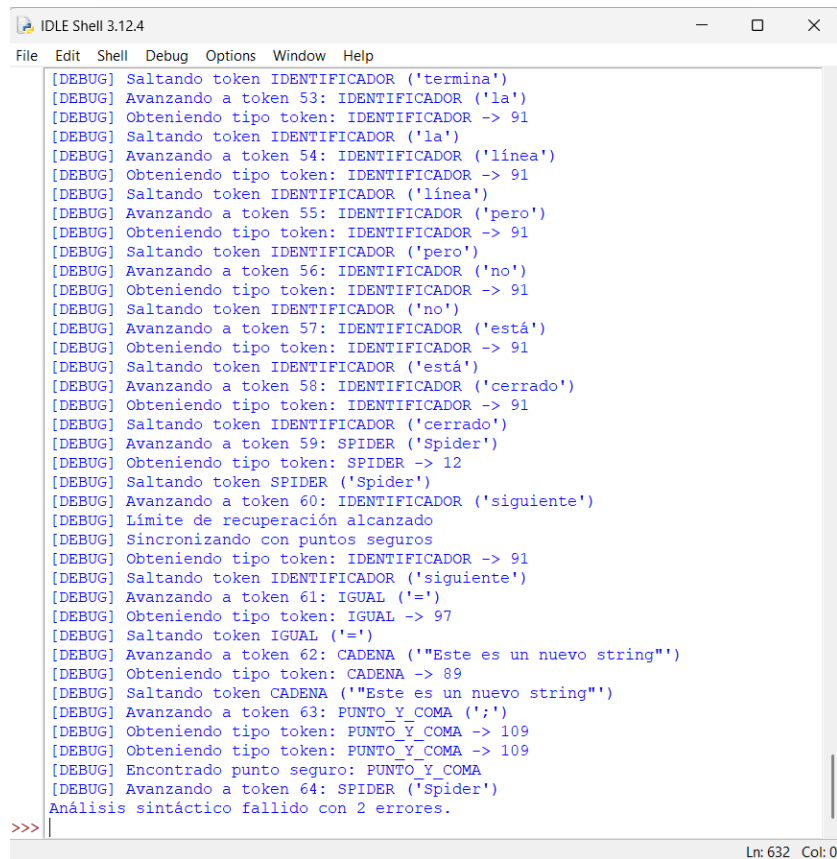
```

Figura 7.3: 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.

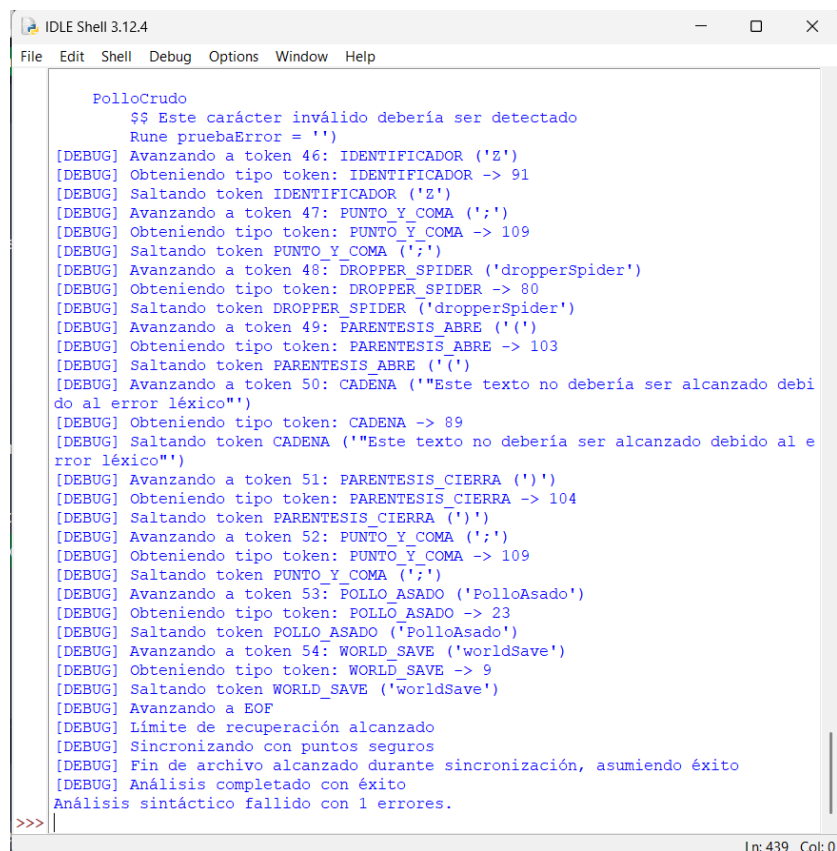
Capturas de pantalla:



```
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.
```

Ln: 632 Col: 0

Figura 7.4: 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 7.5: 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.

Capítulo 8

Documentación del Análisis Semántico

El análisis semántico constituye una de las etapas fundamentales en el proceso de compilación, ya que se encarga de validar el significado de las construcciones sintácticas generadas previamente. A diferencia del análisis sintáctico, que se limita a verificar la estructura gramatical del código fuente, el análisis semántico introduce una capa adicional de verificación lógica que garantiza la coherencia del programa.

Durante esta fase, se revisan aspectos como: la consistencia de tipos de datos, la correcta utilización de identificadores, el cumplimiento de reglas de alcance y la validez de operaciones según el contexto. Esto permite detectar errores que no pueden ser capturados únicamente por la gramática, asegurando así que el programa tenga sentido desde el punto de vista del lenguaje diseñado.

En este capítulo se documentan los principales componentes que conforman el análisis semántico implementado en el compilador del lenguaje ficticio, incluyendo:

- Un **diccionario semántico**, donde se definen las reglas específicas asociadas a cada construcción del lenguaje.
- Una **explicación del código fuente** encargado de realizar estas validaciones durante la etapa semántica.
- La **estructura de la tabla semántica**, utilizada para almacenar y consultar información relevante sobre los identificadores, tipos y valores.

- Un apartado de **prevenciones semánticas**, donde se detallan los chequeos específicos implementados para evitar errores lógicos y semánticos.

A continuación, se presentan estos elementos organizados en secciones, con el fin de ofrecer una guía clara y detallada del funcionamiento interno del análisis semántico del compilador.

8.1. Diccionario Semantico

A continuacion se presenta el diccionario semantico, vital para el funcionamiento del analisis del mismo nombre.

8.1.1. Valor Tipo

Definición

Se implementa la función encargada de realizar el chequeo semántico del valor asociado al tipo de dato, conocido como **valor-tipo**. Esta verificación tiene como propósito asegurar que el valor asignado a una variable sea coherente con el tipo declarado en su definición. Si el valor proporcionado no es válido para el tipo, se asigna automáticamente un valor por defecto que representa un estado neutro o inicial para dicho tipo.

Este chequeo resulta esencial para prevenir errores en tiempo de ejecución y mantener la integridad del sistema de tipos del lenguaje. El lenguaje ficticio en cuestión define un conjunto de tipos primitivos, y para cada uno de ellos se ha establecido una equivalencia por defecto que se aplica en caso de inconsistencia o ausencia de valor explícito.

A continuación, se listan las equivalencias por defecto para cada tipo reconocido:

- **STACK**: Representa un tipo entero. Su valor por defecto es 0, lo cual simboliza una pila vacía o sin elementos.
- **GHASt**: Corresponde a un tipo de número en punto flotante. El valor por defecto es 0.0, utilizado para indicar una cantidad o medida nula.
- **TORCH**: Tipo booleano representado en forma textual. El valor por defecto es ".n", lo que sugiere que una antorcha, por omisión, se encuentra encendida.
- **SPIDER**: Tipo de cadena de texto. El valor por defecto es la cadena vacía , representando un texto no inicializado.
- **RUNE**: Tipo de carácter. El valor por defecto es el carácter nulo \0, indicando la ausencia de un símbolo válido.

Ubicación

Este chequeo semántico se realiza en cada punto del código donde se intenta modificar o asignar un nuevo valor a un símbolo previamente declarado

en la tabla de símbolos semántica. Dicha tabla almacena las definiciones de variables, incluyendo su tipo, nombre, y otras propiedades relevantes.

La verificación de tipo es fundamental para garantizar la consistencia del sistema. Se asegura que el valor que se está asignando a una variable sea compatible con el tipo declarado, y en caso contrario, se aplica el valor por defecto como mecanismo de corrección semántica. Esto permite continuar la ejecución del programa sin comprometer el modelo de tipos ni su semántica.

Este proceso también ayuda a detectar errores comunes de programación, como la asignación de cadenas a variables numéricas o el uso de booleanos en contextos incompatibles, fortaleciendo la robustez del lenguaje.

8.1.2. Chequeo de existencia de identificador

Definición

Cada vez que se detecta un token de tipo `IDENTIFICADOR`, se ejecuta un chequeo semántico para validar su existencia dentro del entorno actual. Este chequeo cumple dos funciones principales: si el identificador aún no ha sido declarado, se procede a crearlo e ingresarlo en la tabla de símbolos; en cambio, si ya existe, se valida que la operación actual sea compatible con su tipo y contexto.

Cuando se pretende modificar el valor de un identificador ya existente, es obligatorio realizar adicionalmente un chequeo de tipo, descrito en la sección correspondiente ([ver: Chequeo de Valor Tipo](#)), con el fin de garantizar que la nueva asignación respete las reglas del sistema de tipos del lenguaje.

Este mecanismo permite mantener la coherencia del entorno semántico, evita redefiniciones ilegales y permite detectar errores de uso indebido de identificadores antes de la generación de código.

Ubicación

Este chequeo debe ejecutarse en cada punto del análisis semántico donde se encuentre un identificador, tanto durante su declaración inicial como al momento de actualizar su valor. La validación asegura la unicidad y coherencia del identificador en el ámbito de ejecución actual.

8.1.3. Chequeo de constantes

Definición

Este chequeo semántico se aplica sobre identificadores del tipo `OBSIDIAN`, los cuales representan constantes dentro del lenguaje. Su objetivo es garan-

tizar que una vez creada, una constante mantenga su valor inmutable a lo largo de toda la ejecución del programa.

Durante la creación de un identificador de tipo **OBSIDIAN**, se verifica que no exista ya en la tabla de símbolos un identificador con el mismo nombre. En caso de encontrarse una colisión, se rechaza la declaración, evitando la sobrescritura de constantes ya establecidas. De este modo, se protege la inmutabilidad semántica de estos elementos y se garantiza la integridad del entorno simbólico.

Este chequeo también cumple una función importante como medida de seguridad: evita que una constante se utilice de forma dinámica como una variable, promoviendo un uso más predecible y seguro del lenguaje.

Ubicación

Este chequeo se aplica cada vez que se intenta declarar un identificador del tipo **OBSIDIAN**. En otras palabras, toda definición de constante debe pasar por este filtro de unicidad e inmutabilidad.

8.1.4. Chequeo de Overflow

Definición

Este chequeo se encarga de prevenir desbordamientos de memoria relacionados con los tipos **STACK**, **SPIDER** y **GHASt**. Dado que el lenguaje objetivo del compilador es un lenguaje ensamblador con registros de tamaño limitado, es crucial verificar que los datos asociados a estas variables no excedan la capacidad de almacenamiento asignada por arquitectura.

La función de este chequeo es analizar el valor asignado o previsto para estas variables, estimar el tamaño en bits o bytes que ocupará en memoria, y compararlo contra los límites definidos por el modelo de ejecución. Si el valor excede el tamaño permitido, se marca como error semántico y se aborta la operación de declaración.

Este mecanismo ayuda a evitar errores que podrían comprometer la ejecución segura del programa generado, tales como corrupciones de pila, lecturas inválidas de memoria o comportamientos impredecibles en bajo nivel.

Ubicación

Este chequeo se activa de manera obligatoria en cada declaración de variables de tipo **STACK**, **SPIDER** o **GHASt**, dado que estos tipos están directamente relacionados con estructuras de datos que ocupan espacio significativo en memoria y requieren verificación explícita de límites.

8.1.5. Chequeo de Pollo

Definición

Este chequeo semántico se encarga de verificar el correcto emparejamiento de los delimitadores estructurales **POLLOCRUDO** y **POLLOASADO**, los cuales actúan como símbolos de apertura y cierre de bloques o módulos dentro del lenguaje. La lógica del chequeo impone que cada aparición de un **POLLOCRUDO** debe ser correspondida eventualmente por un **POLLOASADO**, manteniendo la estructura anidada correctamente balanceada.

Si se encuentra un **POLLOCRUDO**, se inicia un proceso de seguimiento con una lista temporal de tokens y un contador de profundidad. Cada vez que se detecta otro **POLLOCRUDO**, el contador se incrementa. Por cada **POLLOASADO**, se decrementa. El proceso es exitoso si el contador vuelve a cero, lo que indica que todos los bloques han sido correctamente cerrados.

Este chequeo es esencial para la correcta partición modular del programa, garantizando que las unidades de código estén bien definidas y no se generen estructuras abiertas o mal terminadas que comprometan la interpretación o compilación del código.

Ubicación

Se ejecuta inmediatamente después de identificar un token **POLLOCRUDO**. El sistema de análisis busca en adelante el cierre correspondiente, analizando la secuencia de tokens y actualizando el contador de anidación. El chequeo concluye cuando se emparejan todos los delimitadores abiertos, o se lanza un error si no se encuentra cierre válido.

8.1.6. Chequeo de Shelf Size

Definición

Este chequeo valida que, al declarar un identificador del tipo **SHELF** (una estructura de datos similar a un arreglo), se especifiquen correctamente dos elementos fundamentales: la cantidad de elementos que almacenará, y el tipo de dato de esos elementos.

Una vez declarados estos atributos, se verifica que el número de espacios definidos sea coherente con su uso posterior. Declarar una **SHELF** con 10 espacios y luego utilizar 12, implica acceder a memoria no asignada, lo cual es un error semántico grave. Por otro lado, declarar una **SHELF** con más espacios de los que realmente se usarán puede indicar un desperdicio de recursos o una mala planificación del código.

Este chequeo garantiza un uso eficiente y seguro de la memoria, y previene accesos ilegales o inconsistencias en estructuras indexadas.

Ubicación

Este chequeo se realiza cada vez que se declara un identificador de tipo SHELF. Se valida tanto en el momento de la declaración como durante su utilización si se realizan accesos indexados.

8.1.7. Chequeo de tipo de operación

Definición

Este chequeo semántico garantiza que los operadores aritméticos sean utilizados exclusivamente con tipos de datos compatibles. La operación de suma, por ejemplo, debe involucrar operandos del tipo **STACK** (entero) o **GHA**ST (flotante), pero nunca cadenas (**SPIDER**) ni booleanos implícitos.

Este análisis verifica el tipo de los operandos antes de generar el código correspondiente, rechazando combinaciones no válidas o potencialmente peligrosas. Esto asegura la robustez del lenguaje y evita errores en tiempo de ejecución relacionados con operaciones mal tipadas.

En algunos casos, podría permitirse la conversión implícita entre tipos numéricos compatibles (como **STACK** a **GHA**ST), pero nunca entre tipos incompatibles como números y cadenas.

Ubicación

Este chequeo se activa en cada instancia de uso de operadores aritméticos, como **SUMA**, **RESTA**, **MULTIPLICACION** y **DIVISION**, validando el tipo de ambos operandos involucrados.

8.1.8. Chequeo de Uso de Variables

Definición

Este chequeo semántico tiene como objetivo identificar variables declaradas que nunca llegan a utilizarse durante la ejecución del programa. Su propósito principal es facilitar la optimización del código, eliminando declaraciones innecesarias y mejorando la eficiencia tanto en tiempo de compilación como en uso de memoria.

La implementación se realiza a través de un arreglo temporal que registra toda variable declarada a lo largo del análisis. Posteriormente, al finalizar el

análisis semántico, se compara este arreglo contra la tabla de símbolos para verificar que cada variable haya sido utilizada (es decir, accedida o referenciada) al menos una vez. Las variables con valores nulos o sin interacción alguna pueden marcarse como advertencias o errores según las políticas del compilador.

Ubicación

Este chequeo se realiza al finalizar completamente el análisis sintáctico y semántico, en la etapa de verificación global previa a la generación de código.

8.1.9. Chequeo de WorldName

Definición

Este chequeo asegura que la primera palabra efectiva (ignorando espacios, comentarios y tokens irrelevantes) del programa sea el identificador `WORLDNAME`. Este identificador actúa como punto de entrada semántico del programa, y su ausencia o desorden indica un problema estructural grave.

La función de este chequeo es establecer una raíz válida para la ejecución y validación del programa. Además, permite definir un nombre simbólico para el entorno de ejecución, útil para procesos posteriores como serialización, persistencia o vinculación.

Ubicación

Este chequeo se realiza como primer paso del análisis sintáctico. Una vez que se inicia la secuencia de tokens, se busca inmediatamente que el primer token significativo sea `WORLDNAME`.

8.1.10. Chequeo de WorldSave

Definición

Este chequeo garantiza que el programa concluya correctamente con la palabra clave `WORLDSAVE`. Esta palabra simboliza el cierre adecuado del contexto de ejecución y asegura que todos los bloques y estructuras hayan sido finalizados correctamente.

`WORLDSAVE` marca explícitamente el final de la ejecución lógica del programa, permitiendo al compilador validar que no hay instrucciones colgantes, bloques incompletos ni código residual fuera del flujo formal.

Ubicación

Este chequeo se ejecuta como último paso del análisis, tras recorrer todos los tokens del programa. Se espera que el último token significativo sea `WORLDSAVE`, y en caso contrario se lanza un error de sintaxis estructural.

8.1.11. Chequeo de Nombre de Archivo

Definición

Este chequeo valida que los archivos a ser manipulados por el programa tengan la extensión `.txt`, dado que el lenguaje restringe el acceso exclusivamente a archivos de texto plano. Esto previene errores de ejecución al intentar abrir o escribir sobre archivos binarios, no reconocidos por el entorno de ejecución.

Al detectar una operación relacionada con archivos, el sistema valida que el nombre del archivo termine en `.txt`, tanto en declaraciones explícitas como en referencias internas. De no cumplirse esta condición, se reporta un error semántico.

Ubicación

Este chequeo se activa cada vez que se encuentra un identificador `BOOK`, el cual representa una unidad de manejo de archivos dentro del lenguaje. Se analiza el nombre del archivo vinculado al identificador en cuestión.

8.1.12. Chequeo de Igualdad de Operadores

Definición

Este chequeo proporciona soporte semántico a operadores compuestos como `SUMAIGUAL`, `RESTAIGUAL`, `MULTIPLICACIONIGUAL` y `DIVISIONIGUAL`, que combinan asignación con operación aritmética en una sola instrucción. La verificación se encarga de comprobar que:

- El identificador al que se aplica el operador exista y sea modificable.
- El tipo del valor involucrado sea compatible con el tipo de dato original del identificador.
- La operación aritmética implícita sea válida entre los operandos involucrados.

Esto asegura un comportamiento coherente y predecible del programa, y evita errores relacionados con conversiones de tipo implícitas o intentos de operar sobre tipos incompatibles.

Ubicación

Este chequeo se realiza en cada aparición de operadores compuestos como `+=`, `-=`, `*=`, o `/=`, asociando su aplicación a un identificador válido en contexto.

8.1.13. Chequeo de Inicialización de Variables

Definición

Este chequeo cumple la función de detectar si una variable está siendo inicializada por primera vez. Es fundamental para establecer correctamente el tipo de dato que se le debe asociar en la tabla de símbolos y para permitir operaciones futuras que dependan de este tipo, como los operadores compuestos o validaciones semánticas más complejas.

La verificación permite saber si una variable ya posee un valor asignado previamente o si aún está en estado no inicializado. Si no ha sido inicializada, se procede a establecer su valor inicial con una asignación directa, tomando en cuenta el tipo inferido o declarado. En cambio, si ya fue inicializada, se evalúa si el nuevo valor es compatible con su tipo.

Ubicación

Este chequeo se realiza en cada punto del programa donde se intenta asignar un valor a una variable, ya sea en su declaración o en una operación posterior. Es especialmente relevante cuando se efectúan validaciones de operadores que requieren el conocimiento del tipo de dato.

8.1.14. Chequeo de División por Cero

Definición

Este chequeo es una verificación crítica para la estabilidad del programa durante su ejecución. Su objetivo es detectar intentos de realizar una operación de división donde el divisor sea igual a cero, lo cual es un error lógico y provoca fallos en tiempo de ejecución, especialmente a nivel de código ensamblador.

La verificación es simple: antes de ejecutar una operación de división, se comprueba si el valor del divisor es cero. Si es así, se emite un error semántico y se bloquea la generación del código correspondiente.

Ubicación

Este chequeo se ejecuta cada vez que se encuentra un operador de división. Antes de realizar la operación, se analiza el valor del segundo operando para asegurar que no sea cero.

8.1.15. Chequeo de Estructura Target y Hit Miss

Definición

El lenguaje provee una estructura condicional representada por los tokens **TARGET** y **HITMISS**, que funcionan de manera análoga a las estructuras clásicas **if** y **else**. Este chequeo garantiza el correcto orden y lógica de aparición de estos tokens.

Las reglas básicas que se validan son:

- Un **TARGET** siempre debe preceder a cualquier **HITMISS**.
- No pueden existir múltiples **TARGET** consecutivos sin una cláusula de **HITMISS** que cierre el anterior.
- La estructura esperada más óptima es una secuencia alternante: **TARGET** → **HITMISS** → **HITMISS**, permitiendo múltiples condiciones en un mismo bloque.

Este chequeo asegura coherencia estructural y previene ambigüedades en el flujo de control del programa.

Ubicación

Se activa durante el análisis de control de flujo, cada vez que se identifica un token **TARGET** o **HITMISS**, verificando la estructura secuencial completa de estas declaraciones.

8.2. Manejo de Tipo de Datos

A continuación, se describe en detalle el manejo semántico de cada tipo de dato disponible en el lenguaje **Notch Engine**. Esta sección resulta fundamental debido a que cada **IDENTIFICADOR** posee un comportamiento y validaciones específicas dependiendo de su tipo declarado, lo cual condiciona la semántica del programa y su correcta ejecución.

Cabe destacar que este procesamiento está íntimamente ligado a las funciones de verificación de tokens pasados y futuros, previamente descritas. Estas permiten inferir correctamente el contexto en el que se encuentran los identificadores, validar sus estructuras de inicialización y aplicar las reglas semánticas pertinentes.

Cada tipo de dato posee un tratamiento exclusivo, tanto en cuanto a su forma de declaración como a sus restricciones semánticas. A continuación, se detalla el funcionamiento específico para cada tipo:

8.2.1. Manejo de Tipos **STACK**, **GHA**ST, **TORCH**, **SPIDER** y **RUNE**

Los tipos **STACK**, **GHA**ST, **TORCH**, **SPIDER** y **RUNE** son los tipos primitivos fundamentales del lenguaje, y su uso está presente en la mayoría de operaciones básicas. Comparten una estructura de declaración común:

Tipo IDENTIFICADOR = Valor;

El procesamiento comienza tras la validación de la sintaxis de la declaración. Una vez validada, los elementos son enviados a la función semántica **welcomeStack** o a la función inmediata que va a ir, la cual se encarga de analizar en profundidad el contenido de la asignación y registrar la variable en la tabla de símbolos si corresponde.

Los casos más relevantes a considerar durante este análisis son los siguientes:

- **Inicialización Nula:** Ocurre cuando el identificador es declarado sin asignación inmediata de valor. En este caso, se reconoce la variable como declarada pero no inicializada. Esta operación es válida dentro del sistema, y permite postergar la asignación para momentos posteriores en la ejecución, facilitando ciertos patrones de programación estructurada.
- **Instanciación Directa:** Es el caso estándar de inicialización, donde el valor se encuentra disponible en el momento de la declaración. Este

patrón es el más común y recomendable, ya que asegura coherencia semántica inmediata. El sistema valida que el valor sea compatible con el tipo declarado antes de proceder con el almacenamiento.

- **Instanciación Compuesta o Múltiple:** Este es el caso más complejo. Puede involucrar operaciones combinadas entre múltiples identificadores, valores directos o resultados de funciones. Aquí el sistema aplica reglas adicionales para resolver referencias, evaluar expresiones y validar tipos antes de insertar el resultado final. Este tratamiento aplica especialmente a tipos como **STACK** y **GHA**ST, que permiten sobrecarga de operaciones y composición.

8.2.2. Manejo de Tipo BOOK

El tipo **BOOK** introduce una estructura particular, ya que hace uso de una sintaxis basada en paréntesis, dentro de los cuales se debe especificar un archivo y una acción asociada. Por tanto, requiere de una validación más compleja utilizando el análisis de tokens futuros.

```
book IDENTIFICADOR = / .archivos.txt", 'Accion' /;
```

Durante el análisis, se realiza una lectura anticipada de tokens, para capturar elementos clave como el nombre del archivo, la operación a ejecutar, y otros posibles parámetros. Esta estructura obliga a asegurar que todos los elementos dentro de los paréntesis sean correctamente balanceados y estén en orden lógico.

El análisis semántico, en este caso, no solo valida el tipo de datos, sino que también verifica que la acción solicitada sea compatible con el archivo indicado. Una vez verificada la validez de todos los elementos, se autoriza la inserción del identificador tipo **BOOK** en la tabla semántica.

8.2.3. Manejo de Tipo CHEST

El tipo **CHEST** es una estructura semántica que representa una colección de datos en formato de arreglo simple, delimitado por corchetes. A diferencia de otros tipos más rígidos, **CHEST** acepta una variedad de tipos dentro de sus elementos, lo cual lo hace sumamente flexible, pero también propenso a errores semánticos.

```
chest IDENTIFICADOR = : dato1, dato2, dato3;;
```

Dado que este tipo no requiere validación de tipo homogéneo ni cantidad fija de elementos, el énfasis del análisis se centra en la estructura general del arreglo y en asegurar que no existan errores de delimitación, referencias inválidas o tipos mal formateados.

El manejo cuidadoso de **CHEST** es crucial, ya que puede ser utilizado como base para otros tipos más complejos, como **SHELF**.

8.2.4. Manejo de Tipo SHELF

```
Shelf[cantidad] Stack arreglo = [dato1, dato2, dato3];
```

El tipo **SHELF** representa una estructura de datos compuesta que extiende el comportamiento de **CHEST**, pero con requerimientos mucho más estrictos. A diferencia de **CHEST**, **SHELF** exige que todos los elementos dentro del arreglo sean del mismo tipo y que cumplan con una cantidad específica predefinida.

Para validar esto, se aplican varias reglas semánticas previamente definidas, las cuales garantizan homogeneidad y consistencia en la estructura. Además, se emplea una verificación de tipo cruzado, donde si un elemento no cumple con el tipo esperado, se intenta aplicar una corrección automática basada en reglas de conversión implícitas.

Este enfoque asegura robustez semántica y permite que el lenguaje ofrezca una experiencia más tolerante a errores leves, sin comprometer la integridad del programa.

8.2.5. Manejo de Tipo ENTITY

El tipo **ENTITY** simula el comportamiento de una clase u objeto. Su manejo se divide en tres fases principales:

1. **Declaración Estructural:** Se define una entidad con un nombre y una estructura interna. En esta fase no se asignan valores, sino que se especifican los campos o atributos.
2. **Instanciación de Referencia:** Aquí se genera una instancia que referencia la estructura previamente declarada. Esta operación puede hacerse parcialmente, permitiendo que ciertos campos se asignen posteriormente.
3. **Instanciación Directa:** Es una combinación de declaración y asignación, donde se define e inicializa la entidad en una sola línea. Este es el caso más directo, pero requiere validación exhaustiva de tipos, campos y valores asociados.

```
Entity Jugador PolloCrudo Spider nombre; Stack nivel; Ghast salud;  
PolloAsado;
```

```
Entity Jugador jugador = nombre: "Steve", nivel: 1, salud: 20.0;
```

El tipo `ENTITY` es vital para la organización del código y permite aplicar patrones de diseño que requieren encapsulamiento de información.

8.2.6. Manejo de Tipo `SPELL`

El tipo `SPELL` representa funciones puras dentro del lenguaje. Su declaración responde a una estructura específica:

```
Spell nombreFuncion(Tipo :: parametro1, ... ) ->TipoRetorno
```

Durante el análisis semántico, se realiza una validación exhaustiva de los siguientes elementos:

- El nombre de la función debe ser único dentro del espacio de nombres global.
- Cada parámetro debe estar correctamente tipado y separado por comas, con un delimitador de tipo (`::`).
- El tipo de retorno debe coincidir con el resultado devuelto dentro del cuerpo de la función.
- Se asegura que dentro del cuerpo de la función no existan ambigüedades, uso de variables no declaradas o tipos incompatibles.

La tabla semántica almacena esta función como un símbolo especial, con referencia a su tipo, parámetros y tipo de retorno, para permitir llamadas válidas posteriores en el flujo del programa.

8.2.7. Manejo de Tipo `RITUAL`

El tipo `RITUAL` define procedimientos (subrutinas) que no retornan un valor explícito. Su estructura es la siguiente:

```
Ritual nombreProcedimiento(Tipo :: parametro1, ... );
```

En este caso, el análisis semántico asegura lo siguiente:

- El procedimiento no retorna ningún valor, por lo que cualquier intento de utilizar su resultado en una operación generará un error semántico.
- Los parámetros son verificados uno a uno, asegurando su existencia, tipo, y compatibilidad con las llamadas posteriores.
- Se validan posibles efectos secundarios que el procedimiento pueda provocar, incluyendo manipulaciones de variables globales o entidades.

El procedimiento es almacenado en la tabla semántica con una etiqueta especial de tipo nulo (**void**), junto con su nombre y lista de parámetros.

8.3. Explicación del Código Semántico

En esta sección se detallan las funciones y estructuras fundamentales que permiten llevar a cabo el análisis semántico dentro del lenguaje ficticio Notch Engine. Estas herramientas son esenciales para garantizar que el análisis sintáctico no solo reconozca la estructura del código, sino que también valide el significado y coherencia de los elementos declarados y utilizados. A continuación, se presentan las modificaciones aplicadas al parser, así como la implementación de componentes clave como los símbolos, los mecanismos de verificación, el historial semántico y el sistema de manejo de errores.

8.3.1. Modificación del Parser

La modificación del parser es una etapa crítica, en especial en lo que respecta a la función encargada de avanzar al siguiente token. Esta función desempeña un papel determinante, ya que controla el flujo de análisis de cada elemento léxico que compone el programa. Gracias a esta funcionalidad, el compilador adquiere la capacidad de interpretar correctamente cada token, identificando estructuras como identificadores, funciones, operadores y otros elementos clave del lenguaje.

Al integrar lógica semántica en la función de avance, se habilita un control detallado del procesamiento, lo que contribuye significativamente a la robustez y estabilidad del compilador. Esta capacidad se apoya directamente en una serie de funciones auxiliares que se describen a continuación.

8.3.2. Símbolos

Los símbolos constituyen la unidad básica de información para el análisis semántico, ya que representan las entradas que conforman la tabla de símbolos. Esta tabla se abordará con mayor detalle en secciones posteriores.

Cada símbolo almacena suficiente información para identificar unívocamente a cualquier tipo de identificador presente en el código fuente. Entre los datos almacenados se incluyen el tipo de dato, si la variable ha sido inicializada o no, así como la ubicación exacta en el código (línea y columna), lo que resulta sumamente útil para la generación de mensajes de error precisos.

Adicionalmente, los símbolos permiten construir y gestionar eficientemente el diccionario semántico, ya que actúan como unidades encapsuladas de significado. La correcta gestión de esta estructura de datos es esencial para el funcionamiento del análisis semántico.

A continuación se presenta la estructura de datos que representa un símbolo en Notch Engine.

```

class Simbolo:
    def __init__(self, nombre, tipo, categoria, linea,
                  columna, valor=None, otros_atributos=None):
        self.nombre = nombre
        self.tipo = tipo # STACK, SPIDER, GHAST, etc...
        self.categoria = categoria # 'variable',
                                   OBSIDIAN, SPELL, RITUAL
        self.linea = linea # Linea de declaracion
        self.columna = columna # Columna de declaracion
        self.valor = valor # Para constantes o
                           inicializaciones

```

8.3.3. Verificación Pasada de Tokens

La verificación retrospectiva de tokens es una función clave para validar declaraciones, especialmente en lo que respecta a la correcta asociación entre tipos de datos e identificadores. Este mecanismo permite al analizador semántico retroceder en el flujo de tokens para verificar los contextos en los que fueron declarados ciertos elementos, asegurando así la validez de las expresiones o asignaciones posteriores.

Dicha función es empleada de manera recurrente en múltiples verificaciones semánticas, ya que garantiza una interpretación coherente de estructuras que dependen de declaraciones previas.

8.3.4. Verificación Futura de Tokens

Por otro lado, la verificación futura de tokens permite anticiparse a errores que podrían surgir si no se controlara adecuadamente la continuidad de la sintaxis. Esta función es esencial para gestionar correctamente la declaración y uso de variables, estructuras de control, bloques de código, y otros elementos similares.

Por ejemplo, cuando se detecta una apertura de bloque (comúnmente representada por un símbolo especial denominado "POLLO CRUDO", equivalente a un corchete de apertura), es necesario garantizar que dicho bloque sea cerrado correctamente. Para lograr esto sin perder el token actual, se emplean tokens temporales o copias del estado actual, lo cual permite realizar una verificación anticipada sin comprometer el flujo del parser.

Este enfoque resulta significativamente más eficiente que mantener un historial completo de todos los tokens procesados, ya que evita un consumo excesivo de memoria y simplifica la lógica del compilador. Así, se logran

detectar errores de forma inmediata y precisa, mejorando la experiencia de desarrollo y reduciendo la posibilidad de errores en tiempo de ejecución.

8.3.5. Historial Semántico

El historial semántico constituye una herramienta fundamental para visualizar el resultado de las diferentes verificaciones realizadas durante el análisis. Su propósito es ofrecer una trazabilidad clara del proceso de validación semántica, lo cual resulta especialmente útil durante tareas de depuración y control de calidad del código fuente.

Este historial se encuentra dividido en dos categorías principales:

- **Historial Completo:** almacena todas las reglas semánticas evaluadas, sin importar si su resultado fue exitoso o fallido. Esta modalidad representa un registro exhaustivo del comportamiento del compilador y es especialmente útil para realizar auditorías o análisis detallados del código.
- **Historial Negativo:** registra únicamente aquellas reglas que produjeron errores o inconsistencias. Su propósito es facilitar la localización de fallos, simplificando el proceso de depuración y corrección de errores en el código.

Gracias a esta segmentación, el historial semántico permite mantener un control detallado sobre el estado del análisis en todo momento, contribuyendo al desarrollo estable y predecible del proyecto.

8.3.6. Errores Terminales

El compilador de Notch Engine implementa una estrategia de manejo de errores basada en el concepto de recuperación por pánico. Esto implica que, al detectar un error, el sistema intenta continuar la ejecución y minimizar su impacto, evitando así una interrupción abrupta del análisis. Esta técnica se encuentra aplicada en diversas reglas semánticas y gramaticales, como la detección de tipos y sus valores por defecto, o la verificación previa de operaciones válidas.

No obstante, existen ciertos casos en los que se considera necesario interrumpir inmediatamente la compilación. Estos casos corresponden a errores terminales, y son los siguientes:

- **Declaración incorrecta de `WorldName`:** Si el identificador `WorldName` es mal declarado o no es el primer token del programa, el compilador

interrumpe su ejecución de forma inmediata. Esta regla es esencial, ya que `WorldName` marca el inicio del programa y es una parte integral de su estructura sintáctica.

- **Declaración incorrecta de `WorldSave`:** De manera análoga, la ausencia o mal uso de `WorldSave` como cierre del programa también resulta en la terminación inmediata del proceso de análisis. Este token representa el final formal del programa, por lo que su correcta inclusión es obligatoria.
- **Acumulación excesiva de errores:** Si durante el análisis se detectan más de cinco errores gramaticales consecutivos, se considera que el programa es inviable en su forma actual, por lo que el compilador se detiene. Esta política fue adoptada por cuestiones de tiempo y eficiencia, con el objetivo de evitar un análisis inútil que podría generar confusión adicional en el proceso de desarrollo.

Estas reglas son evaluadas de forma inmediata al ingresar al parser, incluso antes de que el análisis semántico comience. Esto permite evitar la ejecución innecesaria de código que, debido a errores fundamentales, nunca podrá ser interpretado correctamente.

8.4. Explicación de la Tabla Semántica

8.4.1. Estructura de la Tabla Semántica

La tabla semántica es una estructura fundamental en el proceso de análisis semántico, ya que permite llevar un control detallado de los identificadores que han sido declarados, así como de sus propiedades asociadas, tales como el tipo de dato, el valor asignado (si lo hubiera), y su contexto de aparición (línea, columna y ámbito).

La implementación de esta tabla se realiza utilizando una estructura de tipo *hash table*, que ofrece un acceso eficiente, en tiempo constante promedio, a los elementos almacenados. Para manejar colisiones, se hace uso de listas enlazadas en cada índice de la tabla.

Este diseño se eligió por su balance entre rendimiento y simplicidad, además de permitir una rápida inserción, búsqueda y eliminación de símbolos, lo cual resulta crucial para el correcto análisis y validación de código en el lenguaje *Notch Engine*.

8.4.2. Análisis de Funcionamiento

Cada vez que se detecta un identificador (de tipo **IDENTIFICADOR**) en el código fuente, se inicia un proceso riguroso de validación semántica. Este proceso implica:

1. Consultar el **diccionario semántico** para verificar si dicho identificador ya ha sido declarado previamente.
2. Determinar el tipo de dato asociado, con base en las reglas de declaración y contexto del lenguaje.
3. Insertar el identificador en la tabla de símbolos si no ha sido declarado antes, o reportar un error semántico si ya existe una entrada previa para el mismo nombre.
4. Enlazar el identificador a un objeto de tipo **Símbolo**, cuya estructura interna fue explicada en la sección anterior. Este objeto contiene toda la información relevante para su análisis posterior.

El uso de una función **hash** personalizada permite distribuir uniformemente los símbolos en la tabla, reduciendo la probabilidad de colisiones. Cuando una colisión ocurre (es decir, cuando dos identificadores distintos generan el

mismo índice hash), estos se almacenan en una lista correspondiente al índice afectado, y se aplica una búsqueda secuencial dentro de dicha lista para localizarlos.

Este mecanismo asegura una alta eficiencia durante la compilación y reduce la complejidad en el manejo de estructuras semánticas, especialmente en programas de gran tamaño o con múltiples declaraciones.

8.4.3. Resultados de la Tabla de Símbolos

El código que se muestra a continuación implementa la clase `TablaSimbolos`, la cual encapsula toda la lógica necesaria para gestionar la estructura hash mencionada. Esta clase está diseñada como un *singleton*, lo que garantiza que solo exista una única instancia de la tabla durante el análisis completo del código fuente, evitando inconsistencias por múltiples referencias.

Listing 8.1: Implementación de la Tabla de Símbolos en Python

```
class TablaSimbolos:
    _instancia = None # Atributo de clase para
                       almacenar la nica instancia

    def __init__(self, tamaño=101):
        if TablaSimbolos._instancia is not None:
            raise Exception("Usa TablaSimbolos._instancia
                             () para obtener la instancia.")
        self.tamaño = tamaño
        self.tabla = [[] for _ in range(tamaño)]

    @classmethod
    def instancia(cls):
        if cls._instancia is None:
            cls._instancia = TablaSimbolos()
        return cls._instancia

    def _hash(self, nombre):
        return sum(ord(c) * (i + 1) for i, c in
                   enumerate(nombre)) % self.tamaño

    def insertar(self, simbolo):
        idx = self._hash(simbolo.nombre)
        for sym in self.tabla[idx]:
            if sym.nombre == simbolo.nombre:
```

```

        raise ValueError(f"Identificador '{
            simbolo.nombre}' ya declarado")
    self.tabla[idx].append(simbolo)

def buscar(self, nombre):
    idx = self._hash(nombre)
    for sym in self.tabla[idx]:
        if sym.nombre == nombre:
            return sym
    return None

def eliminar(self, nombre):
    idx = self._hash(nombre)
    self.tabla[idx] = [sym for sym in self.tabla[idx]
        ] if sym.nombre != nombre]

def imprimir_tabla(self):
    for i, lista in enumerate(self.tabla):
        if lista:
            print(f" ndice {i}:")
            for simbolo in lista:
                print(f" - {simbolo}")

```

En conjunto con las estructuras de símbolos, esta tabla permite implementar reglas semánticas complejas como la verificación de declaraciones duplicadas, asignaciones inválidas, control de tipos, y otros errores semánticos. Asimismo, al ser una estructura centralizada, puede integrarse fácilmente con los demás componentes del compilador, como el historial semántico y los analizadores de contexto.

8.4.4. Ejemplos reales de la tabla semantica

Se hace una referencia importante, los resultados de la tabla semantica estan en la siguiente seccion de resultados semanticos, por lo que se recomienda seguir leyendo el documento. Esto se hace de esta manera para poder tener una mejor visualizacion de los logros realizados.

8.5. Resultados de Semantica

Se debe de saber que al elegir el programa inicialmente se ejecutara y se generara de manera automaticamente los resultados del muro de ladrillos del scanner. Seguidamente al terminar la ejecucion si es que NO hay un error terminal se completara la ejecucion del programa, por lo que es de suma importancia prestar atencion a lo siguiente.

Al terminar la ejecucion del programa se abre un menu con 4 opciones:

1. **Presentacion de la tabla semantica:** Se hace un print de la tabla semantica, se recomienda verla a fondo para hacerse una idea del analisis del programa.
2. **Presentacion del historial semantico:** Se hace una presentacion del historial semantico completo, esto es importante recalcarlo porque cada regla funciona se imprime sin importar que sea una regla positiva o negativa.
3. **Presentacion del historial semantico negativo:** a diferencia del pasado solo se presentan las reglas negativas, incluso se vuelve a recomendar, muchas veces estas reglas no son negativas, pero son posibles llamados de atencion para el compilador, por lo que se debe de tener mucho cuidado a la hora de leerlos e interpretarlos.
4. **Salida:** Al clicar esta seccion se da por finalizado la ejecucion del programa por completo, por lo que se recomienda tener cuidado.

Todo este ejecutable esta hecho para que funcione de manera enloop hasta salirse por completo, ademas se hace de notar que cualquier iteracion que haya una vez llegado a este menu no alterara los resultados, en caso de querer ver cambios se debera de ir a ejecutar el programa otra vez, ademas de que se tiene de referencia que el programa sirve como una referencia al programador, esta forma de debugging es una ayuda tremenda tanto para el programador del lenguaje como para el futuro usuario del mismo. Se espera que para la generacion de codigo estas ayudas puedan darse de manera automatica via comando.

A continuacion se muestran resultados de la prueba *28prueba_1Dsimilares.txt* esta prueba fue ejecutada y se anexan los resultados, se muestra esta prueba por ser larga y robusta.

8.5.1. Resultados de la tabla de hash

```
Información de la tabla de símbolos:
Índice 8:
  - Nombre: SpawnPoint_Main, Tipo: STACK, Categoría: VARIABLE, Línea: 29, Columna: 9, Valor: None
Índice 5:
  - Nombre: onGate, Tipo: STACK, Categoría: VARIABLE, Línea: 86, Columna: 9, Valor: None
Índice 9:
  - Nombre: dropperFunction, Tipo: STACK, Categoría: VARIABLE, Línea: 105, Columna: 9, Valor: None
Índice 11:
  - Nombre: enderpearlTeleport, Tipo: STACK, Categoría: VARIABLE, Línea: 76, Columna: 9, Valor: None
  - Nombre: andOperator, Tipo: STACK, Categoría: VARIABLE, Línea: 84, Columna: 9, Valor: None
Índice 12:
  - Nombre: Bedrock_Type, Tipo: STACK, Categoría: VARIABLE, Línea: 21, Columna: 9, Valor: None
Índice 13:
  - Nombre: CraftingTable2, Tipo: STACK, Categoría: VARIABLE, Línea: 27, Columna: 9, Valor: None
Índice 22:
  - Nombre: PruebaIdentificadoresSimilares, Tipo: WORLD_NAME, Categoría: WORLD_NAME, Línea: 7, Columna: 11, Valor: PruebaIdentificadoresSimilares
Índice 28:
  - Nombre: hopperMinecart, Tipo: STACK, Categoría: VARIABLE, Línea: 103, Columna: 9, Valor: None
Índice 31:
  - Nombre: RecipeBook, Tipo: STACK, Categoría: VARIABLE, Línea: 25, Columna: 9, Valor: None
```

Figura 8.1: Tabla 1

```
Índice 31:
  - Nombre: RecipeBook, Tipo: STACK, Categoría: VARIABLE, Línea: 25, Columna: 9, Valor: None
  - Nombre: enderpearlItem, Tipo: STACK, Categoría: VARIABLE, Línea: 65, Columna: 9, Valor: None
Índice 35:
  - Nombre: InventorySlot, Tipo: STACK, Categoría: VARIABLE, Línea: 23, Columna: 9, Valor: None
Índice 37:
  - Nombre: Spell_Cast, Tipo: STACK, Categoría: VARIABLE, Línea: 70, Columna: 9, Valor: None
Índice 39:
  - Nombre: SoulSandBlock, Tipo: STACK, Categoría: VARIABLE, Línea: 80, Columna: 9, Valor: None
Índice 43:
  - Nombre: dropper, Tipo: STACK, Categoría: VARIABLE, Línea: 104, Columna: 9, Valor: None
Índice 46:
  - Nombre: BedrockType, Tipo: SPIDER, Categoría: OBSIDIAN, Línea: 12, Columna: 19, Valor: "Hard Stone"
Índice 48:
  - Nombre: MagmaBlock, Tipo: STACK, Categoría: VARIABLE, Línea: 82, Columna: 9, Valor: None
Índice 58:
  - Nombre: WorldNameTest, Tipo: STACK, Categoría: OBSIDIAN, Línea: 11, Columna: 18, Valor: 100
Índice 62:
  - Nombre: WorldName2, Tipo: STACK, Categoría: VARIABLE, Línea: 19, Columna: 9, Valor: None
  - Nombre: polloAsadoExtra, Tipo: STACK, Categoría: VARIABLE, Línea: 43, Columna: 9, Valor: None
Fin del archivo
```

Figura 8.2: Tabla 2

```
Índice 65:
  - Nombre: xorCalculation, Tipo: STACK, Categoría: VARIABLE, Línea: 90, Columna: 9, Valor: None
  - Nombre: onSwitch, Tipo: TORCH, Categoría: VARIABLE, Línea: 97, Columna: 9, Valor: None
Índice 66:
  - Nombre: InventoryFull, Tipo: TORCH, Categoría: OBSIDIAN, Línea: 13, Columna: 18, Valor: On
Índice 67:
  - Nombre: worldNameGenerator, Tipo: SPIDER, Categoría: FUNCTION, Línea: 109, Columna: 9, Valor: {'parametros': [], 'tipo_retorno': 'SPIDER', 'tiene_impl_...
Índice 74:
  - Nombre: notOperator, Tipo: STACK, Categoría: VARIABLE, Línea: 88, Columna: 9, Valor: None
Índice 77:
  - Nombre: hopper, Tipo: STACK, Categoría: VARIABLE, Línea: 102, Columna: 9, Valor: None
Índice 87:
  - Nombre: respawnPoint, Tipo: STACK, Categoría: VARIABLE, Línea: 74, Columna: 9, Valor: None
  - Nombre: craftingTableMaker, Tipo: VOID, Categoría: PROTOTIPO_PROC, Línea: 110, Columna: 10, Valor: {'parametros': [], 'tipo_retorno': 'VOID', 'tiene_...
Índice 89:
  - Nombre: isEqual, Tipo: STACK, Categoría: VARIABLE, Línea: 93, Columna: 9, Valor: None
  - Nombre: generatedName, Tipo: SPIDER, Categoría: VARIABLE, Línea: 150, Columna: 12, Valor: worldNameGenerator
Índice 90:
  - Nombre: ender_pearl, Tipo: STACK, Categoría: VARIABLE, Línea: 75, Columna: 9, Valor: None
Índice 99:
  - Nombre: RitualCeremony, Tipo: STACK, Categoría: VARIABLE, Línea: 72, Columna: 9, Valor: None
```

Figura 8.3: Tabla 3

8.5.2. Resultados de historial semantico

```
Historial Semantico:
-----
01. REGLA SEMANTICA 030: POLLOCRUDDO abierto en línea 41, columna 9 (contexto: bloque_general, nivel: 1)
02. REGLA SEMANTICA 030: POLLOASADO cierra bloque de línea 41 en línea 42, columna 9 (contexto: bloque_general)
03. REGLA SEMANTICA 030: POLLOCRUDDO abierto en línea 110, columna 3 (contexto: funcion, nivel: 1)
04. REGLA SEMANTICA 030: POLLOASADO cierra bloque de línea 110 en línea 116, columna 3 (contexto: funcion)
05. REGLA SEMANTICA 030: POLLOCRUDDO abierto en línea 119, columna 3 (contexto: bloque_general, nivel: 1)
06. REGLA SEMANTICA 030: POLLOASADO cierra bloque de línea 119 en línea 125, columna 3 (contexto: bloque_general)
07. REGLA SEMANTICA 030: POLLOCRUDDO abierto en línea 129, columna 3 (contexto: bloque_general, nivel: 1)
08. REGLA SEMANTICA 030: POLLOCRUDDO abierto en línea 144, columna 5 (contexto: estructura_control, nivel: 2)
09. REGLA SEMANTICA 030: POLLOASADO cierra bloque de línea 144 en línea 147, columna 5 (contexto: estructura_control)
10. REGLA SEMANTICA 030: POLLOASADO cierra bloque de línea 129 en línea 152, columna 3 (contexto: bloque_general)
11. REGLA SEMANTICA 030: Todos Los bloques POLLOCRUDDO/POLLOASADO están correctamente balanceados
12. REGLA SEMANTICA 030: Encontrados 5 POLLOCRUDDO y 5 POLLOASADO. Bloques correctos: 5. Máxima anidación: 2. Balance CORRECTO.
13. REGLA SEMANTICA 008: El primer Token del Programa es 'WorldName' con tipo 'WORLD_NAME', se PROCEDE con la ejecucion.
14. REGLA SEMANTICA 009: El ultimo Token del Programa es 'worldSave' con tipo 'WORLD_SAVE', se PROCEDE con la ejecucion.
15. REGLA SEMANTICA 001: EL IDENTIFICADOR de nombre 'PruebaIdentificadoresSimilares' NO existe, se procede a crear
16. REGLA SEMANTICA 012: La variable 'PruebaIdentificadoresSimilares' (tipo: WORLD_NAME) ha sido correctamente inicializada con el valor 'PruebaIdentificadoresSimilares'
17. REGLA SEMANTICA 001: EL IDENTIFICADOR de nombre 'WorldNameTest' NO existe, se procede a crear
18. REGLA SEMANTICA 002: EL IDENTIFICADOR de nombre 'WorldNameTest' NO existe, se procede a crear el OBSIDIAN
19. REGLA SEMANTICA 000: Valor '100' (str) convertido a STACK + 100
```

Figura 8.4: Historial Semantico Completo 1

```
20. REGLA SEMANTICA 012: La variable 'WorldNameTest' (tipo: STACK) ha sido correctamente inicializada con el valor '100'. Línea: 11, Columna: 18.
21. REGLA SEMANTICA 001: EL IDENTIFICADOR de nombre 'BedrockType' NO existe, se procede a crear
22. REGLA SEMANTICA 002: EL IDENTIFICADOR de nombre 'BedrockType' NO existe, se procede a crear el OBSIDIAN
23. REGLA SEMANTICA 000: Valor '"Hard Stone"' convertido a SPIDER → '"Hard Stone"'
24. REGLA SEMANTICA 012: La variable 'BedrockType' (tipo: SPIDER) ha sido correctamente inicializada con el valor '"Hard Stone"'. Línea: 12, Columna: 19.
25. REGLA SEMANTICA 001: EL IDENTIFICADOR de nombre 'InventoryFull' NO existe, se procede a crear
26. REGLA SEMANTICA 002: EL IDENTIFICADOR de nombre 'InventoryFull' NO existe, se procede a crear el OBSIDIAN
27. REGLA SEMANTICA 000: Valor '0n' (str) convertido a TORCH + 0n
28. REGLA SEMANTICA 012: La variable 'InventoryFull' (tipo: TORCH) ha sido correctamente inicializada con el valor '0n'. Línea: 13, Columna: 18.
29. REGLA SEMANTICA 001: EL IDENTIFICADOR de nombre 'WorldName2' NO existe, se procede a crear
30. REGLA SEMANTICA 012: La variable 'WorldName2' (tipo: STACK) fue declarada pero NO ha sido inicializada. Línea: 19, Columna: 9.
31. REGLA SEMANTICA 007: Variable 'WorldName2' declarada, contador inicializado en 0.
32. REGLA SEMANTICA 001: EL IDENTIFICADOR de nombre 'Bedrock_Type' NO existe, se procede a crear
33. REGLA SEMANTICA 012: La variable 'Bedrock_Type' (tipo: STACK) fue declarada pero NO ha sido inicializada. Línea: 21, Columna: 9.
34. REGLA SEMANTICA 007: Variable 'Bedrock_Type' declarada, contador inicializado en 0.
35. REGLA SEMANTICA 001: EL IDENTIFICADOR de nombre 'InventorySlot' NO existe, se procede a crear
36. REGLA SEMANTICA 012: La variable 'InventorySlot' (tipo: STACK) fue declarada pero NO ha sido inicializada. Línea: 23, Columna: 9.
37. REGLA SEMANTICA 007: Variable 'InventorySlot' declarada, contador inicializado en 0.
38. REGLA SEMANTICA 001: EL IDENTIFICADOR de nombre 'RecipeBook' NO existe, se procede a crear
39. REGLA SEMANTICA 012: La variable 'RecipeBook' (tipo: STACK) fue declarada pero NO ha sido inicializada. Línea: 25, Columna: 9.
40. REGLA SEMANTICA 007: Variable 'RecipeBook' declarada, contador inicializado en 0.
```

Figura 8.5: Historial Semantico Completo 2

```
41. REGLA SEMANTICA 001: EL IDENTIFICADOR de nombre 'CraftingTable2' NO existe, se procede a crear
42. REGLA SEMANTICA 012: La variable 'CraftingTable2' (tipo: STACK) fue declarada pero NO ha sido inicializada. Línea: 27, Columna: 9.
43. REGLA SEMANTICA 007: Variable 'CraftingTable2' declarada, contador inicializado en 0.
44. REGLA SEMANTICA 001: EL IDENTIFICADOR de nombre 'SpawnPoint_Main' NO existe, se procede a crear
45. REGLA SEMANTICA 012: La variable 'SpawnPoint_Main' (tipo: STACK) fue declarada pero NO ha sido inicializada. Línea: 29, Columna: 9.
46. REGLA SEMANTICA 007: Variable 'SpawnPoint_Main' declarada, contador inicializado en 0.
47. REGLA SEMANTICA 001: EL IDENTIFICADOR de nombre 'polloAsadoExtra' NO existe, se procede a crear
48. REGLA SEMANTICA 012: La variable 'polloAsadoExtra' (tipo: STACK) fue declarada pero NO ha sido inicializada. Línea: 43, Columna: 9.
49. REGLA SEMANTICA 007: Variable 'polloAsadoExtra' declarada, contador inicializado en 0.
50. REGLA SEMANTICA 035: TARGET detectado en línea 47, columna 9 (ID: 0, contexto: target_anidado, nivel: 1)
51. REGLA SEMANTICA 035: HIT procesado en línea 48, columna 9 para TARGET (ID: 0, línea 47)
52. REGLA SEMANTICA 035: Estructura TARGET (ID: 0) cerrada implícitamente por PUNTO_Y_COMA en línea 48
53. REGLA SEMANTICA 035: ERROR - MISS sin TARGET correspondiente en línea 49, columna 9
54. REGLA SEMANTICA 001: EL IDENTIFICADOR de nombre 'enderpearlItem' NO existe, se procede a crear
55. REGLA SEMANTICA 012: La variable 'enderpearlItem' (tipo: STACK) fue declarada pero NO ha sido inicializada. Línea: 65, Columna: 9.
56. REGLA SEMANTICA 007: Variable 'enderpearlItem' declarada, contador inicializado en 0.
57. REGLA SEMANTICA 001: EL IDENTIFICADOR de nombre 'Spell_Cast' NO existe, se procede a crear
58. REGLA SEMANTICA 012: La variable 'Spell_Cast' (tipo: STACK) fue declarada pero NO ha sido inicializada. Línea: 70, Columna: 9.
59. REGLA SEMANTICA 007: Variable 'Spell_Cast' declarada, contador inicializado en 0.
60. REGLA SEMANTICA 001: EL IDENTIFICADOR de nombre 'RitualCeremony' NO existe, se procede a crear
61. REGLA SEMANTICA 012: La variable 'RitualCeremony' (tipo: STACK) fue declarada pero NO ha sido inicializada. Línea: 72, Columna: 9.
```

Figura 8.6: Historial Semantico Completo 3

```

62. REGLA SEMANTICA 007: Variable 'RitualCeremony' declarada, contador inicializado en 0.
63. REGLA SEMANTICA 001: El IDENTIFICADOR de nombre 'respawnPoint' NO existe, se procede a crear
64. REGLA SEMANTICA 012: La variable 'respawnPoint' (tipo: STACK) fue declarada pero NO ha sido inicializada. Línea: 74, Columna: 9.
65. REGLA SEMANTICA 007: Variable 'respawnPoint' declarada, contador inicializado en 0.
66. REGLA SEMANTICA 001: El IDENTIFICADOR de nombre 'ender_pearl' NO existe, se procede a crear
67. REGLA SEMANTICA 012: La variable 'ender_pearl' (tipo: STACK) fue declarada pero NO ha sido inicializada. Línea: 75, Columna: 9.
68. REGLA SEMANTICA 007: Variable 'ender_pearl' declarada, contador inicializado en 0.
69. REGLA SEMANTICA 001: El IDENTIFICADOR de nombre 'enderpearlTeleport' NO existe, se procede a crear
70. REGLA SEMANTICA 012: La variable 'enderpearlTeleport' (tipo: STACK) fue declarada pero NO ha sido inicializada. Línea: 76, Columna: 9.
71. REGLA SEMANTICA 007: Variable 'enderpearlTeleport' declarada, contador inicializado en 0.
72. REGLA SEMANTICA 001: El IDENTIFICADOR de nombre 'SoulSandBlock' NO existe, se procede a crear
73. REGLA SEMANTICA 012: La variable 'SoulSandBlock' (tipo: STACK) fue declarada pero NO ha sido inicializada. Línea: 80, Columna: 9.
74. REGLA SEMANTICA 007: Variable 'SoulSandBlock' declarada, contador inicializado en 0.
75. REGLA SEMANTICA 001: El IDENTIFICADOR de nombre 'MagmaBlock' NO existe, se procede a crear
76. REGLA SEMANTICA 012: La variable 'MagmaBlock' (tipo: STACK) fue declarada pero NO ha sido inicializada. Línea: 82, Columna: 9.
77. REGLA SEMANTICA 007: Variable 'MagmaBlock' declarada, contador inicializado en 0.
78. REGLA SEMANTICA 001: El IDENTIFICADOR de nombre 'andOperator' NO existe, se procede a crear
79. REGLA SEMANTICA 012: La variable 'andOperator' (tipo: STACK) fue declarada pero NO ha sido inicializada. Línea: 84, Columna: 9.
80. REGLA SEMANTICA 007: Variable 'andOperator' declarada, contador inicializado en 0.
81. REGLA SEMANTICA 001: El IDENTIFICADOR de nombre 'orGate' NO existe, se procede a crear
82. REGLA SEMANTICA 012: La variable 'orGate' (tipo: STACK) fue declarada pero NO ha sido inicializada. Línea: 86, Columna: 9.

```

Figura 8.7: Historial Semantico Completo 4

```

83. REGLA SEMANTICA 007: Variable 'orGate' declarada, contador inicializado en 0.
84. REGLA SEMANTICA 001: El IDENTIFICADOR de nombre 'notOperator' NO existe, se procede a crear
85. REGLA SEMANTICA 012: La variable 'notOperator' (tipo: STACK) fue declarada pero NO ha sido inicializada. Línea: 88, Columna: 9.
86. REGLA SEMANTICA 007: Variable 'notOperator' declarada, contador inicializado en 0.
87. REGLA SEMANTICA 001: El IDENTIFICADOR de nombre 'xorCalculation' NO existe, se procede a crear
88. REGLA SEMANTICA 012: La variable 'xorCalculation' (tipo: STACK) fue declarada pero NO ha sido inicializada. Línea: 90, Columna: 9.
89. REGLA SEMANTICA 007: Variable 'xorCalculation' declarada, contador inicializado en 0.
90. REGLA SEMANTICA 001: El IDENTIFICADOR de nombre 'isEqual' NO existe, se procede a crear
91. REGLA SEMANTICA 012: La variable 'isEqual' (tipo: STACK) fue declarada pero NO ha sido inicializada. Línea: 93, Columna: 9.
92. REGLA SEMANTICA 007: Variable 'isEqual' declarada, contador inicializado en 0.
93. REGLA SEMANTICA 001: El IDENTIFICADOR de nombre 'onSwitch' NO existe, se procede a crear
94. REGLA SEMANTICA 012: La variable 'onSwitch' (tipo: TORCH) fue declarada pero NO ha sido inicializada. Línea: 97, Columna: 9.
95. REGLA SEMANTICA 001: El IDENTIFICADOR de nombre 'offState' NO existe, se procede a crear
96. REGLA SEMANTICA 012: La variable 'offState' (tipo: TORCH) fue declarada pero NO ha sido inicializada. Línea: 99, Columna: 9.
97. REGLA SEMANTICA 001: El IDENTIFICADOR de nombre 'hopper' NO existe, se procede a crear
98. REGLA SEMANTICA 012: La variable 'hopper' (tipo: STACK) fue declarada pero NO ha sido inicializada. Línea: 102, Columna: 9.
99. REGLA SEMANTICA 007: Variable 'hopper' declarada, contador inicializado en 0.
100. REGLA SEMANTICA 001: El IDENTIFICADOR de nombre 'hopperMinecart' NO existe, se procede a crear
101. REGLA SEMANTICA 012: La variable 'hopperMinecart' (tipo: STACK) fue declarada pero NO ha sido inicializada. Línea: 103, Columna: 9.
102. REGLA SEMANTICA 007: Variable 'hopperMinecart' declarada, contador inicializado en 0.
103. REGLA SEMANTICA 001: El IDENTIFICADOR de nombre 'dropper' NO existe, se procede a crear

```

Figura 8.8: Historial Semantico Completo 5

```

104. REGLA SEMANTICA 012: La variable 'dropper' (tipo: STACK) fue declarada pero NO ha sido inicializada. Línea: 104, Columna: 9.
105. REGLA SEMANTICA 007: Variable 'dropper' declarada, contador inicializado en 0.
106. REGLA SEMANTICA 001: El IDENTIFICADOR de nombre 'dropperFunction' NO existe, se procede a crear
107. REGLA SEMANTICA 012: La variable 'dropperFunction' (tipo: STACK) fue declarada pero NO ha sido inicializada. Línea: 105, Columna: 9.
108. REGLA SEMANTICA 007: Variable 'dropperFunction' declarada, contador inicializado en 0.
109. REGLA SEMANTICA 001: El IDENTIFICADOR de nombre 'worldNameGenerator' NO existe, se procede a crear
110. REGLA SEMANTICA 012: La variable 'worldNameGenerator' (tipo: SPIDER) ha sido correctamente inicializada con el valor '{'parametros': [], 'tipo_retorno': 'craftingTableMaker'}.
111. REGLA SEMANTICA 001: El IDENTIFICADOR de nombre 'craftingTableMaker' NO existe, se procede a crear
112. REGLA SEMANTICA 012: La variable 'craftingTableMaker' (tipo: VOID) ha sido correctamente inicializada con el valor '{'parametros': [], 'tipo_retorno': 'craftingTableMaker'}.
113. REGLA SEMANTICA 030: POLICRUDDO abierto en línea 129, columna 3 (contexto: bloque_general, nivel: 1)
114. REGLA SEMANTICA 035: TARGET detectado en línea 143, columna 5 (ID: 1, contexto: target_anidado, nivel: 1)
115. REGLA SEMANTICA 035: HIT procesado en línea 143, columna 37 para TARGET (ID: 1, línea 143)
116. REGLA SEMANTICA 035: Estructura TARGET (ID: 1) cerrada implícitamente por PUNTO_Y_COMA en línea 145
117. REGLA SEMANTICA 001: El IDENTIFICADOR de nombre 'generatedName' NO existe, se procede a crear
118. REGLA SEMANTICA 008: Valor 'worldNameGenerator' convertido a SPIDER → 'worldNameGenerator'
119. REGLA SEMANTICA 012: String de longitud 18 está dentro del límite para SPIDER para variable 'generatedName'
120. REGLA SEMANTICA 012: La variable 'generatedName' (tipo: SPIDER) ha sido correctamente inicializada con el valor 'worldNameGenerator'. Línea: 150, Columna: 9.
121. REGLA SEMANTICA 001: El IDENTIFICADOR de nombre 'worldNameGenerator' ya existe
122. REGLA SEMANTICA 001: El IDENTIFICADOR de nombre 'ender_pearl' ya existe
123. REGLA SEMANTICA 001: El IDENTIFICADOR de nombre 'craftingTableMaker' ya existe
124. REGLA SEMANTICA 035: Validación final TARGET/HIT/MISS - TARGET detectados: 2, Completados: 2, Con MISS: 0, Con errores: 0, Anidación máxima: 0 - 1 E

```

Figura 8.9: Historial Semantico Completo 6

8.5.3. Resultados de historial semantico negativo

```
-----
01. REGLA SEMANTICA 012: La variable 'WorldName2' (tipo: STACK) fue declarada pero NO ha sido inicializada. Línea: 19, Columna: 9.
02. REGLA SEMANTICA 012: La variable 'Bedrock_Type' (tipo: STACK) fue declarada pero NO ha sido inicializada. Línea: 21, Columna: 9.
03. REGLA SEMANTICA 012: La variable 'InventorySlot' (tipo: STACK) fue declarada pero NO ha sido inicializada. Línea: 23, Columna: 9.
04. REGLA SEMANTICA 012: La variable 'RecipeBook' (tipo: STACK) fue declarada pero NO ha sido inicializada. Línea: 25, Columna: 9.
05. REGLA SEMANTICA 012: La variable 'CraftingTable2' (tipo: STACK) fue declarada pero NO ha sido inicializada. Línea: 27, Columna: 9.
06. REGLA SEMANTICA 012: La variable 'SpawnPoint_Main' (tipo: STACK) fue declarada pero NO ha sido inicializada. Línea: 29, Columna: 9.
07. REGLA SEMANTICA 012: La variable 'polloAsadoExtra' (tipo: STACK) fue declarada pero NO ha sido inicializada. Línea: 43, Columna: 9.
08. REGLA SEMANTICA 012: La variable 'enderpearlItem' (tipo: STACK) fue declarada pero NO ha sido inicializada. Línea: 65, Columna: 9.
09. REGLA SEMANTICA 012: La variable 'Spell_Cast' (tipo: STACK) fue declarada pero NO ha sido inicializada. Línea: 70, Columna: 9.
10. REGLA SEMANTICA 012: La variable 'RitualCeremony' (tipo: STACK) fue declarada pero NO ha sido inicializada. Línea: 72, Columna: 9.
11. REGLA SEMANTICA 012: La variable 'respawnPoint' (tipo: STACK) fue declarada pero NO ha sido inicializada. Línea: 74, Columna: 9.
12. REGLA SEMANTICA 012: La variable 'ender_pearl' (tipo: STACK) fue declarada pero NO ha sido inicializada. Línea: 75, Columna: 9.
13. REGLA SEMANTICA 012: La variable 'enderpearlTeleport' (tipo: STACK) fue declarada pero NO ha sido inicializada. Línea: 76, Columna: 9.
14. REGLA SEMANTICA 012: La variable 'SoulSandBlock' (tipo: STACK) fue declarada pero NO ha sido inicializada. Línea: 80, Columna: 9.
15. REGLA SEMANTICA 012: La variable 'MagmaBlock' (tipo: STACK) fue declarada pero NO ha sido inicializada. Línea: 82, Columna: 9.
16. REGLA SEMANTICA 012: La variable 'andOperator' (tipo: STACK) fue declarada pero NO ha sido inicializada. Línea: 84, Columna: 9.
17. REGLA SEMANTICA 012: La variable 'orGate' (tipo: STACK) fue declarada pero NO ha sido inicializada. Línea: 86, Columna: 9.
18. REGLA SEMANTICA 012: La variable 'notOperator' (tipo: STACK) fue declarada pero NO ha sido inicializada. Línea: 88, Columna: 9.
19. REGLA SEMANTICA 012: La variable 'xorCalculation' (tipo: STACK) fue declarada pero NO ha sido inicializada. Línea: 90, Columna: 9.
20. REGLA SEMANTICA 012: La variable 'isEqual' (tipo: STACK) fue declarada pero NO ha sido inicializada. Línea: 93, Columna: 9.
```

Figura 8.10: Historial Semantico Completo NEGATIVA 1

Capítulo 9

Generación de Código

9.1. Introducción

La generación de código constituye la etapa final y culminante del proceso de compilación del lenguaje Notch Engine. En esta fase crítica, el compilador transforma las estructuras de datos internas y la representación semántica del programa fuente en instrucciones ejecutables de código ensamblador TASM, completando así el ciclo completo de traducción desde el código fuente de alto nivel hasta el código objeto ejecutable.

Esta etapa representa la materialización práctica de todo el análisis previo realizado durante las fases de análisis léxico, sintáctico y semántico. La información recopilada en la tabla de símbolos semánticos, las verificaciones de tipos realizadas, y la estructura sintáctica validada del programa sirven como fundamento para la generación de código eficiente y correctamente estructurado.

9.1.1. Objetivos de la Generación de Código

El generador de código del compilador Notch Engine persigue varios objetivos fundamentales que garantizan la correcta traducción y ejecución del programa objetivo:

Correctitud Semántica

El código generado debe preservar fielmente la semántica original del programa fuente. Cada construcción del lenguaje Notch Engine debe traducirse a una secuencia equivalente de instrucciones en ensamblador que mantenga el comportamiento esperado. Esto incluye la correcta implementación de operaciones aritméticas, estructuras de control, manejo de variables y gestión de

memoria.

Gestión de Recursos

El generador debe administrar eficientemente los recursos del sistema objetivo, incluyendo registros del procesador, segmentos de memoria (datos y código), y la pila del sistema. Se implementa un control estricto para prevenir el desbordamiento de segmentos, constituyendo un error irrecoverable que debe detectarse durante la generación.

Interfaz con Runtime Library

Todas las operaciones complejas desarrolladas en etapas anteriores se integran en una biblioteca de tiempo de ejecución unificada (Runtime Library). Esta biblioteca proporciona las funcionalidades básicas del lenguaje como operaciones de entrada/salida, manipulación de cadenas, y gestión de estructuras de datos complejas.

9.1.2. Arquitectura del Generador

El generador de código se estructura siguiendo un diseño modular que separa claramente las responsabilidades y facilita el mantenimiento y extensión del sistema:

Tabla de Símbolos de Generación

Se mantiene una tabla de símbolos específica para la generación de código, independiente de la tabla semántica utilizada en etapas anteriores. Esta separación permite mantener información específica de generación como direcciones de memoria, offsets de variables, y etiquetas de código sin contaminar el análisis semántico.

Manejo de Segmentos

El generador implementa un sistema de gestión de segmentos que controla automáticamente el uso del espacio en los segmentos de datos y código. Se mantiene un registro preciso del espacio utilizado y disponible, generando errores irrecoverables cuando se alcancen los límites de capacidad.

Generación de Portadas

Cada programa generado incluye automáticamente una portada de identificación que contiene información del proyecto, autores, y metadatos relevantes. Esta característica asegura la trazabilidad y documentación adecuada del código generado.

9.1.3. Estrategia de Traducción

La traducción del código fuente a ensamblador sigue una estrategia dirigida por la sintaxis, donde cada construcción sintáctica del lenguaje Notch Engine tiene asociada una rutina específica de generación de código. Este enfoque garantiza la consistencia y facilita la verificación de la correctitud de la traducción.

Traducción de Expresiones

Las expresiones aritméticas y lógicas se traducen utilizando una estrategia de evaluación en pila, donde los operandos se cargan en la pila del sistema y los operadores se implementan mediante llamadas a rutinas especializadas de la Runtime Library.

Estructuras de Control

Las estructuras de control como `TARGET/HITMISS` (equivalentes a `if/else`) y los bucles se implementan mediante etiquetas y saltos condicionales, manteniendo la semántica original del flujo de control del programa fuente.

Gestión de Variables

Las variables se mapean a ubicaciones específicas en memoria, considerando su tipo de dato, alcance, y tiempo de vida. Se implementa un sistema de direccionamiento que permite el acceso eficiente tanto a variables locales como globales.

9.1.4. Manejo de Errores en Generación

Durante la fase de generación de código, todos los errores se consideran irreversibles y provocan la terminación inmediata del proceso de compilación mediante una estrategia de pánico. Esta decisión de diseño reconoce que los errores en esta etapa típicamente indican problemas fundamentales que no pueden ser corregidos automáticamente.

Los errores de tiempo de ejecución del código generado siguen la misma filosofía, terminando la ejecución del programa de manera controlada y proporcionando información diagnóstica útil para la depuración.

9.1.5. Integración con Etapas Anteriores

El generador de código se integra seamlessly con las etapas previas del compilador, utilizando la información recopilada durante el análisis léxico, sintáctico y semántico. La tabla de símbolos semánticos proporciona información de tipos y alcances, mientras que el árbol sintáctico abstracto guía la estructura de la generación de código.

Esta integración permite aprovechar al máximo el trabajo realizado en etapas anteriores, minimizando la duplicación de esfuerzo y asegurando la consistencia global del compilador.

El resultado final de esta etapa es un programa ejecutable en ensamblador TASM que preserva fielmente la semántica del programa original en Notch Engine, listo para su ensamblado y ejecución en el sistema objetivo.

9.2. Análisis de Código

9.2.1. Chequeo de Variables No Inicializadas

Definición

Este chequeo se encarga de verificar si una variable está siendo utilizada antes de haber sido inicializada con un valor válido. Durante el análisis de flujo de datos, se rastrea el estado de inicialización de cada variable declarada, manteniendo un registro de aquellas que han recibido un valor y aquellas que permanecen en estado no inicializado.

Cuando se detecta el uso de una variable no inicializada, el sistema puede aplicar diferentes estrategias: emitir una advertencia y asignar un valor por defecto según el tipo de dato, o generar un error que detenga la compilación dependiendo de las políticas de seguridad establecidas.

Este mecanismo previene comportamientos impredecibles en tiempo de ejecución y ayuda a identificar errores lógicos en el código fuente, especialmente en casos donde el programador asume incorrectamente que una variable ha sido inicializada previamente.

Ubicación

Se ejecuta cada vez que se referencia una variable en una expresión, asignación o como parámetro de función. El análisis se realiza antes de la generación de código para garantizar que todas las variables utilizadas tengan valores válidos.

9.2.2. Chequeo de Código Muerto

Definición

El análisis de código muerto (dead code) identifica segmentos de código que nunca serán ejecutados durante la ejecución normal del programa. Esto incluye instrucciones que aparecen después de declaraciones de terminación como `WORLDSAVE`, declaraciones de retorno incondicionales, o bloques que se encuentran en ramas de control inalcanzables.

La detección de código muerto es importante tanto para la optimización del programa final como para alertar al programador sobre posibles errores lógicos. El código inalcanzable puede indicar condiciones mal estructuradas, bucles con condiciones incorrectas o declaraciones de control de flujo mal ubicadas.

El análisis utiliza técnicas de análisis de flujo de control para determinar qué instrucciones pueden ser alcanzadas desde el punto de entrada del programa, marcando como código muerto aquellas que no pueden ser ejecutadas bajo ninguna circunstancia.

Ubicación

Se realiza durante el análisis de flujo de control del programa, después de construir el grafo de flujo de control y antes de la generación de código optimizado.

9.2.3. Chequeo de Tipos en Operaciones Binarias

Definición

Este chequeo valida la compatibilidad de tipos en operaciones binarias, asegurando que los operandos involucrados en operaciones aritméticas, lógicas o de comparación sean semánticamente compatibles. El sistema verifica que no se realicen operaciones inválidas como sumar una cadena (`SPIDER`) con un entero (`STACK`).

Para cada operador binario, existe un conjunto de reglas que define qué combinaciones de tipos son válidas. Por ejemplo, las operaciones aritméticas

requieren operandos numéricos (**STACK** o **GHA**ST), mientras que las operaciones de concatenación pueden requerir tipos específicos como cadenas.

En casos donde los tipos no son directamente compatibles pero existe una conversión implícita válida, el sistema puede aplicar la conversión automáticamente o emitir una advertencia según las políticas de tipado establecidas.

Ubicación

Se activa al encontrar operadores aritméticos (+, −, *, /), operadores lógicos (&&, ||), operadores de comparación (==, !=, <, >) y otros operadores binarios durante el análisis semántico.

9.2.4. Chequeo de Límites en SHELF

Definición

Este chequeo específico para estructuras de tipo **SHELF** (equivalentes a arreglos) verifica que todos los accesos indexados se mantengan dentro de los límites declarados de la estructura. Se valida tanto que el índice no sea negativo como que no exceda el tamaño máximo definido durante la declaración.

El análisis puede ser estático (cuando los índices son constantes conocidas en tiempo de compilación) o dinámico (insertando verificaciones en tiempo de ejecución para índices calculados). Para índices estáticos, se puede determinar con certeza si el acceso es válido, mientras que para índices dinámicos se generan verificaciones de rango.

La violación de límites en arreglos es una fuente común de errores graves que pueden llevar a corrupción de memoria, accesos a datos no válidos o comportamientos impredecibles del programa.

Ubicación

Se ejecuta cada vez que se detecta un acceso indexado a una estructura **SHELF**, tanto en operaciones de lectura como de escritura.

9.2.5. Chequeo de Constantes Reasignadas

Definición

Este chequeo garantiza la inmutabilidad de las constantes declaradas con el tipo **OBSIDIAN**. Una vez que una constante ha sido inicializada con un valor, cualquier intento posterior de modificar dicho valor debe ser rechazado como un error semántico.

El sistema mantiene un registro de todas las constantes declaradas y sus valores asociados. Durante el análisis, cada operación de asignación se verifica contra este registro para asegurar que no se esté intentando modificar una constante existente.

Este mecanismo es fundamental para mantener las garantías de inmutabilidad que proporcionan las constantes, permitiendo optimizaciones del compilador y asegurando comportamientos predecibles en el código.

Ubicación

Se activa en cada operación de asignación, verificando si el identificador de destino corresponde a una constante previamente declarada.

9.2.6. Chequeo de Argumentos en Llamadas a Funciones

Definición

Este chequeo valida que las llamadas a funciones o procedimientos se realicen con el número correcto de argumentos y que cada argumento sea del tipo esperado según la signatura de la función. Se verifica tanto la cantidad de parámetros como la compatibilidad de tipos entre los argumentos proporcionados y los parámetros formales.

Para cada función declarada, se almacena su signatura completa incluyendo el número de parámetros, sus tipos y el tipo de retorno. Durante una llamada a función, esta información se utiliza para validar que la invocación sea correcta.

En caso de discrepancias, el sistema puede permitir conversiones implícitas de tipos compatibles o rechazar la llamada completamente dependiendo de la severidad de la incompatibilidad.

Ubicación

Se ejecuta en cada punto donde se realiza una llamada a función o procedimiento, antes de generar el código de invocación.

9.2.7. Chequeo de Retorno en Funciones

Definición

Este chequeo asegura que las funciones que declaran un tipo de retorno específico efectivamente retornen un valor del tipo correcto en todos los ca-

minos de ejecución posibles. Se verifica que no existan rutas de ejecución que terminen sin una declaración de retorno válida.

El análisis examina todos los caminos posibles a través del cuerpo de la función, identificando aquellos que no terminan en una declaración de retorno explícita. Para funciones con tipo de retorno, esto constituye un error semántico que debe ser corregido.

También se valida que el tipo del valor retornado sea compatible con el tipo declarado en la signature de la función, aplicando las mismas reglas de compatibilidad de tipos utilizadas en otras verificaciones.

Ubicación

Se realiza al finalizar el análisis del cuerpo de cada función, verificando todos los caminos de ejecución posibles.

9.2.8. Chequeo de Archivos No Cerrados

Definición

Este chequeo identifica recursos de archivo (tipo `BOOK`) que han sido abiertos durante la ejecución del programa pero no han sido explícitamente cerrados antes de la terminación. La gestión incorrecta de recursos de archivo puede llevar a fugas de recursos y problemas de rendimiento.

El sistema mantiene un registro de todos los archivos abiertos y sus estados correspondientes. Al final del análisis del programa, se verifica que todos los archivos abiertos hayan sido cerrados apropiadamente mediante las operaciones correspondientes.

Esta verificación es especialmente importante en lenguajes que no cuentan con recolección automática de basura para recursos del sistema, donde la responsabilidad de liberar recursos recae completamente en el programador.

Ubicación

Se ejecuta como parte del análisis final del programa, antes de la declaración `WORLDSAVE`, verificando el estado de todos los recursos de archivo.

9.2.9. Chequeo de Bucles Infinitos

Definición

Este análisis identifica bucles que potencialmente nunca terminarán debido a condiciones que permanecen constantes o variables de control que nunca

se modifican dentro del cuerpo del bucle. Se examina si las variables utilizadas en la condición del bucle son modificadas en algún punto del cuerpo del bucle.

El análisis puede detectar bucles infinitos obvios (como `while(true)`) así como casos más sutiles donde las variables de control no se actualizan de manera que permita que la condición del bucle eventualmente se vuelva falsa.

Aunque algunos bucles infinitos pueden ser intencionales (como bucles principales de programas), en la mayoría de casos representan errores lógicos que deben ser señalados al programador.

Ubicación

Se activa al analizar estructuras de control repetitivas como **TARGET** con condiciones de bucle, examinando el cuerpo del bucle en busca de modificaciones a las variables de la condición.

9.2.10. Chequeo de Shadowing de Variables

Definición Este chequeo detecta situaciones donde una variable declarada en un ámbito interno oculta (shadow) a una variable con el mismo nombre en un ámbito externo. Aunque sintácticamente válido, el shadowing puede crear confusión y errores lógicos difíciles de detectar.

El análisis mantiene un registro de los ámbitos anidados y las variables declaradas en cada uno. Cuando se detecta una nueva declaración de variable, se verifica si existe una variable con el mismo nombre en algún ámbito externo accesible.

Dependiendo de las políticas del compilador, el shadowing puede generar una advertencia para alertar al programador sobre la potencial ambigüedad, sin necesariamente impedir la compilación.

Ubicación

Se ejecuta cada vez que se declara una nueva variable, verificando contra todas las variables declaradas en ámbitos externos accesibles.

9.2.11. Chequeo de Conversiones Implícitas Peligrosas

Definición

Este chequeo identifica conversiones automáticas de tipos que pueden resultar en pérdida de datos o comportamientos inesperados. Por ejemplo,

la conversión de un número en punto flotante (**GHOST**) a un entero (**STACK**) resulta en la pérdida de la parte fraccionaria.

Se analizan todas las asignaciones y operaciones que involucran tipos diferentes pero compatibles, evaluando si la conversión implícita puede resultar en pérdida de información o precisión. El sistema puede emitir advertencias para alertar al programador sobre estas situaciones.

Las conversiones consideradas peligrosas incluyen truncamiento de números flotantes, desbordamiento de enteros, y conversiones entre tipos de diferentes rangos de valores.

Ubicación

Se activa en asignaciones, operaciones aritméticas y paso de parámetros donde se detecten tipos diferentes que requieren conversión implícita.

9.2.12. Chequeo de Recursos sin Liberar

Definición

Este análisis identifica recursos del sistema (como memoria asignada dinámicamente para estructuras **SHELF**) que han sido asignados pero no han sido explícitamente liberados antes de que salgan de su ámbito de validez.

El sistema rastrea las operaciones de asignación y liberación de recursos, manteniendo un registro del estado de cada recurso. Al finalizar el análisis de un ámbito, se verifica que todos los recursos asignados en ese ámbito hayan sido apropiadamente liberados.

Esta verificación es crucial para prevenir fugas de memoria y otros problemas relacionados con la gestión de recursos que pueden degradar el rendimiento del programa o agotar los recursos del sistema. **Ubicación** Se realiza al finalizar el análisis de cada ámbito donde se han asignado recursos, y como parte del análisis global antes de la terminación del programa.

9.2.13. Chequeo de Pre/Post Condiciones

Definición

Este chequeo valida que se cumplan las condiciones previas necesarias antes de ejecutar operaciones críticas. El ejemplo más común es la verificación de división por cero, pero puede extenderse a otras condiciones como acceso a punteros nulos o validación de rangos.

Para operaciones que tienen prerequisites específicos, el análisis examina los valores de los operandos (cuando son conocidos en tiempo de compila-

ción) o inserta verificaciones en tiempo de ejecución para garantizar que las condiciones se cumplan.

Este tipo de verificación es fundamental para la robustez del programa, previniendo errores en tiempo de ejecución que podrían resultar en terminación anormal del programa o comportamientos indefinidos.

Ubicación

Se ejecuta antes de operaciones críticas como divisiones, accesos a memoria, y otras operaciones que tienen precondiciones específicas.

9.2.14. Chequeo de Consistencia en `WORLDNAME/-WORLDSAVE`

Definición

Este chequeo verifica la estructura fundamental del programa, asegurando que comience con `WORLDNAME` y termine con `WORLDSAVE`. Esta verificación es crítica para la correcta interpretación del programa y establece los límites claros del código ejecutable.

Se valida no solo la presencia de estos tokens sino también su posición correcta dentro de la estructura del programa. `WORLDNAME` debe ser el primer token significativo, mientras que `WORLDSAVE` debe ser el último, sin código ejecutable después de él.

La ausencia o mal posicionamiento de estos elementos estructurales indica un error fundamental en la organización del programa que debe ser corregido antes de proceder con la compilación.

Ubicación

Se realiza como parte del análisis estructural inicial y final del programa, verificando la presencia y posición correcta de los delimitadores principales.

9.2.15. Chequeo de Rendimiento

Definición

Este análisis identifica patrones de código que pueden resultar en bajo rendimiento, como bucles anidados con operaciones costosas, accesos repetitivos a estructuras de datos complejas, o algoritmos con complejidad innecesariamente alta.

El sistema examina la estructura del código en busca de patrones conocidos que típicamente resultan en problemas de rendimiento. Esto incluye operaciones costosas dentro de bucles, accesos no optimizados a arreglos, y patrones de recursión ineficientes.

Aunque estos patrones no constituyen errores semánticos, su identificación temprana permite al programador optimizar el código antes de que se conviertan en problemas de rendimiento significativos en la ejecución.

Ubicación

Se ejecuta como parte del análisis de optimización, examinando estructuras repetitivas, llamadas a funciones, y patrones de acceso a datos para identificar oportunidades de mejora. ““

9.3. Runtime Library

9.3.1. Introducción a la Runtime Library

La Runtime Library del compilador Notch Engine constituye una biblioteca unificada de rutinas en tiempo de ejecución que implementa todas las operaciones fundamentales del lenguaje. Esta biblioteca representa la culminación del trabajo realizado en etapas anteriores, integrando las operaciones aritméticas, lógicas, de comparación, entrada/salida y manipulación de datos desarrolladas durante el análisis léxico, sintáctico y semántico.

La biblioteca se genera automáticamente como un archivo ensamblador independiente (`runtime_library.asm`) que contiene todas las rutinas necesarias para soportar la ejecución de programas Notch Engine compilados.

9.3.2. Arquitectura de la Runtime Library

Estructura Modular

La Runtime Library se organiza en módulos funcionales que agrupan operaciones relacionadas:

- **Operaciones Aritméticas:** Suma, resta, multiplicación, división y módulo para tipos `STACK`
- **Operaciones de Comparación:** Todas las comparaciones relacionales y de igualdad
- **Operaciones Lógicas:** AND, OR, NOT y XOR para tipos `TORCH`

- **Operaciones de Incremento/Decremento:** Implementación de `soulsand` y `netherrack`
- **Entrada/Salida:** Rutinas para lectura y escritura de diferentes tipos de datos
- **Manipulación de Cadenas:** Operaciones básicas con tipos `SPIDER`

Convención de Llamadas

Todas las rutinas de la Runtime Library siguen una convención de llamadas estándar:

- Los parámetros se pasan a través de la pila del sistema
- El resultado se retorna en el registro `AX` para operaciones que retornan valores
- Se preservan los registros `BP` y otros registros críticos
- Se realiza limpieza automática de la pila al retornar

9.3.3. Implementación de Operaciones

Operaciones Aritméticas Enteras

Las operaciones aritméticas para el tipo `STACK` se implementan como rutinas optimizadas que manejan casos especiales:

SUMAR_ENTEROS: Implementa la operación `:+` sumando dos valores enteros pasados por pila.

RESTAR_ENTEROS: Implementa la operación `:-` restando el segundo operando del primero.

MULTIPLICAR_ENTEROS: Implementa la operación `:*` utilizando la instrucción `IMUL` para multiplicación con signo.

DIVIDIR_ENTEROS: Implementa la operación `:/` con verificación automática de división por cero, retornando 0 en caso de error.

MODULO_ENTEROS: Nueva rutina que implementa la operación módulo `:%`, retornando el resto de la división entera.

Operaciones de Incremento y Decremento

INCREMENTAR_ENTERO: Implementa la operación `soulsand`, incrementando en 1 el valor pasado como parámetro.

DECREMENTAR_ENTERO: Implementa la operación `netherrack`, decrementando en 1 el valor pasado como parámetro.

Operaciones de Comparación

Se implementan todas las operaciones de comparación necesarias para estructuras de control:

COMPARAR_IGUAL: Implementa `is`, comparando si dos valores son iguales.

COMPARAR_DIFERENTE: Implementa `isNot`, comparando si dos valores son diferentes.

COMPARAR_MAYOR: Implementa `>`, comparando si el primer valor es mayor que el segundo.

COMPARAR_MENOR: Implementa `<`, comparando si el primer valor es menor que el segundo.

COMPARAR_MAYOR_IGUAL: Implementa `>=`, comparando si el primer valor es mayor o igual que el segundo.

COMPARAR_MENOR_IGUAL: Implementa `<=`, comparando si el primer valor es menor o igual que el segundo.

Operaciones Lógicas Booleanas

Para el manejo del tipo `TORCH` se implementan operaciones lógicas estándar:

AND_LOGICO: Implementa la operación AND lógica entre dos valores booleanos.

OR_LOGICO: Implementa la operación OR lógica entre dos valores booleanos.

NOT_LOGICO: Implementa la negación lógica de un valor booleano.

XOR_LOGICO: Implementa la operación XOR lógica entre dos valores booleanos.

Operaciones de Entrada/Salida

La biblioteca incluye rutinas especializadas para cada tipo de dato:

LEER_ENTERO: Lee un valor entero desde la entrada estándar y lo retorna como tipo `STACK`.

MOSTRAR_ENTERO: Muestra un valor entero en la salida estándar con formato apropiado.

LEER_CHARACTER: Lee un carácter individual desde la entrada estándar.

MOSTRAR_CHARACTER: Muestra un carácter en la salida estándar.

LEER_BOOLEANO: Lee un valor booleano, aceptando representaciones textuales como `.°N/OFF`.

MOSTRAR_BOOLEANO: Muestra un valor booleano como `.°N.°°FF` según corresponda.

9.3.4. Operaciones de Cadenas

Manipulación Básica

Para el tipo SPIDER se implementan operaciones fundamentales:

CONCATENAR_STRINGS: Une dos cadenas de caracteres creando una nueva cadena resultado.

LONGITUD_STRING: Calcula y retorna la longitud en caracteres de una cadena.

9.3.5. Generación Automática

Proceso de Creación

La Runtime Library se genera automáticamente mediante la clase **GeneradorConRuntime** que:

1. Crea el archivo `runtime_library.asm` con todas las rutinas necesarias
2. Incluye comentarios descriptivos y metadatos de generación
3. Implementa manejo de errores en operaciones críticas como división por cero
4. Optimiza el código generado para eficiencia en tiempo de ejecución

Integración con el Código Generado

El generador de código principal integra la Runtime Library mediante:

- Inclusión automática del archivo mediante directiva `INCLUDE`
- Generación de llamadas apropiadas a las rutinas según las operaciones detectadas
- Manejo automático de parámetros y valores de retorno
- Verificación de compatibilidad de tipos antes de generar llamadas

9.3.6. Manejo de Errores en Runtime

Errores Irrecuperables

La Runtime Library implementa detección y manejo de errores críticos:

- **División por cero:** Se detecta automáticamente y se retorna un valor seguro (0)
- **Desbordamiento de enteros:** Se detecta en operaciones que pueden causar overflow
- **Acceso fuera de rango:** Para operaciones con estructuras SHELF

Estrategia de Recuperación

Cuando se detecta un error en runtime:

1. Se registra el error interno para diagnóstico
2. Se retorna un valor por defecto seguro según el tipo de operación
3. Se continúa la ejecución para permitir terminación controlada
4. Se proporciona información de depuración cuando es posible

9.3.7. Optimizaciones Implementadas

Eficiencia en Llamadas

Las rutinas están optimizadas para:

- Minimizar el uso de registros y preservar estado
- Realizar limpieza automática de pila
- Utilizar instrucciones nativas del procesador cuando es posible
- Evitar operaciones redundantes o costosas

Gestión de Memoria

La biblioteca implementa gestión eficiente de memoria:

- Uso mínimo de memoria para variables temporales
- Reutilización de registros para operaciones múltiples
- Limpieza automática de recursos temporales
- Prevención de fragmentación de memoria

9.3.8. Compatibilidad y Portabilidad

Compatibilidad con TASM

Toda la Runtime Library está diseñada para ser compatible con el ensamblador TASM:

- Utiliza sintaxis estándar de TASM para todas las instrucciones
- Implementa convenciones de llamada compatibles
- Usa directivas de ensamblador apropiadas
- Mantiene compatibilidad con diferentes versiones de TASM

Extensibilidad

La arquitectura permite extensión futura:

- Adición de nuevas rutinas sin modificar las existentes
- Soporte para tipos de datos adicionales
- Integración de optimizaciones específicas de hardware
- Personalización de comportamiento según necesidades específicas

La Runtime Library representa así la base sólida sobre la cual se ejecutan todos los programas compilados con Notch Engine, proporcionando funcionalidad robusta, eficiente y extensible para todas las operaciones fundamentales del lenguaje.

9.4. Ejecución de Programa

9.4.1. Introducción al Proceso de Ejecución

La ejecución de programas Notch Engine compilados representa la culminación de todo el proceso de compilación. Una vez que el código fuente ha sido procesado a través de las etapas de análisis léxico, sintáctico, semántico y generación de código, el resultado es un programa ejecutable en ensamblador TASM que debe ser ensamblado y ejecutado en el sistema objetivo.

Esta etapa final involucra la coordinación entre el código generado por el compilador, la Runtime Library, y el sistema operativo objetivo para producir la ejecución correcta del programa original.

9.4.2. Arquitectura de Ejecución

Segmentación de Memoria

El programa ejecutable generado utiliza una arquitectura de memoria segmentada que divide el espacio de direcciones en tres segmentos principales:

Segmento de Pila (STACK): Reserva 256 palabras (512 bytes) para el manejo de la pila del sistema. Este segmento gestiona las llamadas a procedimientos, parámetros de funciones, variables locales temporales y el estado de ejecución durante las llamadas a la Runtime Library.

Segmento de Datos (DATA): Contiene todas las variables declaradas en el programa, constantes, cadenas literales y estructuras de datos como arreglos (SHELF). Este segmento se organiza secuencialmente según el orden de declaración de variables.

Segmento de Código (CODE): Alberga todas las instrucciones ejecutables del programa, incluyendo el procedimiento principal (MAIN), rutinas auxiliares y puntos de entrada y salida del sistema.

Inicialización del Sistema

Al iniciar la ejecución, el programa realiza una secuencia de inicialización crítica:

1. **Configuración de Segmentos:** Se establece la configuración inicial mediante **ASSUME** que define la asociación entre registros de segmento y segmentos de memoria.
2. **Inicialización de Registros:** El registro DS (Data Segment) se inicializa para apuntar al segmento de datos, permitiendo el acceso correcto a variables y constantes.
3. **Configuración de Pila:** El sistema configura automáticamente el segmento de pila y establece los punteros SP (Stack Pointer) y BP (Base Pointer) en posiciones iniciales válidas.

9.4.3. Modelo de Ejecución

Flujo de Control Principal

La ejecución comienza en el procedimiento **MAIN**, que actúa como punto de entrada único del programa. Desde este punto:

- Se ejecutan secuencialmente las instrucciones generadas a partir del código fuente

- Las estructuras de control (**TARGET/HITMISS**) se implementan mediante saltos condicionales y etiquetas
- Los bucles se ejecutan mediante combinaciones de comparaciones y saltos
- Las operaciones complejas se delegan a la Runtime Library mediante llamadas a procedimientos

Gestión de Variables y Memoria

Durante la ejecución, el sistema gestiona diferentes tipos de variables según su naturaleza:

Variables Simples (STACK, GHOST, TORCH): Se almacenan directamente en el segmento de datos con acceso directo mediante direccionamiento absoluto.

Constantes (OBSIDIAN): Se almacenan en el segmento de datos pero con protección implícita contra modificación (verificada durante la compilación).

Cadenas (SPIDER): Se almacenan como secuencias de bytes terminadas en '\$' siguiendo las convenciones de DOS/TASM.

Arreglos (SHELF): Se implementan como bloques contiguos de memoria con cálculo de direcciones basado en índices y tamaño de elementos.

9.4.4. Integración con Runtime Library

Mecanismo de Llamadas

Todas las operaciones complejas del lenguaje Notch Engine se ejecutan mediante llamadas a rutinas de la Runtime Library:

1. **Preparación de Parámetros:** Los operandos se colocan en la pila en orden inverso (último parámetro primero).
2. **Invocación de Rutina:** Se ejecuta la instrucción **CALL** correspondiente a la operación deseada.
3. **Recuperación de Resultado:** El resultado se obtiene del registro **AX** tras la ejecución de la rutina.
4. **Limpieza de Pila:** La rutina se encarga automáticamente de limpiar los parámetros de la pila.

Operaciones Típicas

Durante la ejecución se realizan llamadas frecuentes a rutinas como:

- SUMAR_ENTEROS, RESTAR_ENTEROS, etc. para operaciones aritméticas
- COMPARAR_IGUAL, COMPARAR_MAYOR, etc. para evaluación de condiciones
- MOSTRAR_ENTERO, LEER_CARACTER, etc. para operaciones de entrada/salida
- AND_LOGICO, OR_LOGICO, etc. for operaciones lógicas booleanas

9.4.5. Manejo de Entrada/Salida

Operaciones dropperTorch y dropperSpider

Las operaciones de salida del lenguaje Notch Engine se traducen a llamadas específicas:

dropperTorch: Se traduce a la rutina `MOSTRAR_ENTERO` o `MOSTRAR_BOOLEANO` según el tipo de dato, mostrando el valor en la consola del sistema.

dropperSpider: Se traduce a `MOSTRAR_CARACTER` o rutinas de manejo de cadenas para mostrar texto en la salida estándar.

Operaciones de Lectura

Las operaciones implícitas de lectura se manejan mediante:

- `LEER_ENTERO` para captura de valores numéricos
- `LEER_CARACTER` para captura de caracteres individuales
- `LEER_BOOLEANO` para captura de valores lógicos con conversión automática

9.4.6. Control de Flujo en Ejecución

Estructuras Condicionales

Las estructuras `TARGET/HITMISS` se ejecutan mediante:

1. Evaluación de la condición mediante llamadas a rutinas de comparación
2. Salto condicional a la etiqueta correspondiente según el resultado
3. Ejecución del bloque `HITMISS` (then) o continuar al bloque alternativo
4. Confluencia en punto común tras completar la estructura condicional

Estructuras Repetitivas

Los bucles implícitos en las estructuras **TARGET** se ejecutan mediante:

- Establecimiento de etiqueta de inicio del bucle
- Evaluación de condición de continuación
- Salto condicional de salida del bucle si la condición es falsa
- Ejecución del cuerpo del bucle
- Salto incondicional de regreso al inicio para reevaluación

9.4.7. Terminación del Programa

Secuencia de Terminación Normal

Un programa Notch Engine termina normalmente mediante la secuencia:

1. Ejecución de la declaración **WORLDSAVE** traducida como punto de terminación
2. Ejecución de la secuencia de terminación del sistema operativo (**MOV AH, 4Ch; INT 21h**)
3. Liberación automática de recursos del sistema (memoria, archivos, etc.)
4. Retorno del control al sistema operativo con código de salida 0

Manejo de Errores en Tiempo de Ejecución

Cuando ocurren errores durante la ejecución:

- **Errores de Runtime Library:** Se manejan internamente retornando valores seguros
- **Errores del Sistema:** Se propagan al sistema operativo con códigos de error apropiados
- **Errores de Acceso a Memoria:** Generan terminación controlada con información diagnóstica
- **Desbordamiento de Pila:** Se detecta y maneja con terminación segura

9.4.8. Optimizaciones de Ejecución

Eficiencia en Runtime

Durante la ejecución, se aplican varias optimizaciones:

- **Reutilización de Registros:** Los valores se mantienen en registros cuando es posible para evitar accesos a memoria
- **Eliminación de Operaciones Redundantes:** El código generado evita cálculos innecesarios
- **Optimización de Saltos:** Se minimizan los saltos condicionales e incondicionales
- **Uso Eficiente de la Pila:** Se minimiza el uso de la pila para operaciones temporales

Gestión de Recursos

El sistema de ejecución implementa gestión eficiente de recursos:

- Liberación automática de memoria temporal tras operaciones complejas
- Reutilización de espacios de memoria para variables de ámbito limitado
- Optimización del uso de registros del procesador
- Minimización de accesos a memoria externa

9.4.9. Depuración y Diagnóstico

Información de Depuración

El código generado incluye información útil para depuración:

- Comentarios que indican la operación original del código fuente
- Etiquetas descriptivas que facilitan el seguimiento del flujo de ejecución
- Preservación de nombres de variables originales cuando es posible
- Información de línea y contexto en caso de errores

Herramientas de Diagnóstico

Para facilitar la depuración y análisis:

- El generador produce código con estructura clara y comentarios descriptivos
- Se mantiene trazabilidad entre código fuente y código generado
- Se proporcionan mensajes de error descriptivos en caso de fallas
- Se incluyen verificaciones de sanidad en puntos críticos

9.4.10. Compatibilidad y Portabilidad

Compatibilidad con Sistemas DOS

El código generado es totalmente compatible con:

- Sistemas DOS y compatibles (incluyendo emuladores modernos)
- Ensamblador TASM en diferentes versiones
- Arquitecturas x86 de 16 bits
- Convenciones estándar de llamadas del sistema DOS

Portabilidad del Código

La arquitectura de ejecución permite:

- Ejecución en diferentes entornos compatibles con x86
- Adaptación a diferentes tamaños de memoria disponible
- Personalización de la Runtime Library para necesidades específicas
- Extensión futura para arquitecturas adicionales

El modelo de ejecución implementado garantiza así que los programas Notch Engine compilados se ejecuten de manera correcta, eficiente y robusta en el sistema objetivo, preservando fielmente la semántica del programa original mientras aprovechan las capacidades del hardware y sistema operativo subyacente.

9.5. Prueba de Implementacion

Para comprobar la prueba de implementacion se va a usar de referencia el Examen 3 del curso de Introduccion a la Programacion (IC1802) en el primer semestre del 2023 a cargo del profesor Aurelio Sanabria.

En la carpeta de pruebas viene anexado el PDF con el enunciado completo, a continuacion se anexan imagenes de las preguntas y los codigos que dan solucion a las mismas.

9.5.1. Pregunta 1

Pregunta 1. Matriz simétrica

Una matriz simétrica es una matriz cuadrada que es **idéntica** a su transpuesta

Puesto matemáticamente: $A = A^T$

Puesto ejemplificamente:

$$A = \begin{vmatrix} 1 & 2 & 4 & 7 \\ 2 & 3 & 5 & 8 \\ 4 & 5 & 6 & 9 \\ 7 & 8 & 9 & 0 \end{vmatrix} \quad A^T = \begin{vmatrix} 1 & 2 & 4 & 7 \\ 2 & 3 & 5 & 8 \\ 4 & 5 & 6 & 9 \\ 7 & 8 & 9 & 0 \end{vmatrix}$$

Programe una función que permita determinar si una matriz es una matriz simétrica

La función debe llamarse de la siguiente forma: `es_matriz_simetrica(matriz) → True/False`

donde:

1. **matriz** es una matriz de acuerdo a la definición vista en clase (sino 'Error01')
2. **matriz** es una matriz cuadrada (sino 'Error02')
3. La **salida** debe ser un valor de verdad *True* si es simétrica, *False* en caso contrario.

Figura 9.1: Pregunta 1

Codigo de Notch Engine

```
WorldName MatrizSimetrica:
```

```
Inventory
```

```
Shelf[10][10] Stack matrizEjemplo;
```

```
CraftingTable
```

```

Spell es_matriz_simetrica(Shelf :: matriz) -> Torch
PolloCrudo
    $$ Verificar que el parámetro es una matriz (Error01)
    target chunk(matriz) >> Stack < 2 craft hit
    PolloCrudo
        respawn "Error01";
    PolloAsado

    $$ Obtener dimensiones de la matriz
    Stack filas = #(matriz);
    Stack columnas = #(matriz[0]);

    $$ Verificar que es cuadrada (Error02)
    target filas isNot columnas craft hit
    PolloCrudo
        respawn "Error02";
    PolloAsado

    $$ Comparar matriz con su transpuesta
    walk i set 0 to filas - 1 craft
    PolloCrudo
        walk j set 0 to columnas - 1 craft
        PolloCrudo
            target matriz[i][j] isNot matriz[j][i] craft hit
            PolloCrudo
                respawn Off;
            PolloAsado
        PolloAsado
    PolloAsado

    $$ Si llegamos aquí, es simétrica
    respawn On;
PolloAsado

SpawnPoint
    PolloCrudo
        $$ Ejemplo de uso
        $$ Matriz simétrica 3x3
        Shelf[3][3] Stack matriz1 = [
            [1, 2, 3],
            [2, 4, 5],

```

```

        [3, 5, 6]
];

$$ Matriz no simétrica 3x3
Shelf[3][3] Stack matriz2 = [
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9]
];

$$ No es matriz (Error01)
Stack noMatriz = 5;

$$ No es cuadrada (Error02)
Shelf[2][3] Stack matrizRectangular = [
    [1, 2, 3],
    [4, 5, 6]
];

$$ Pruebas
dropperSpider("Matriz 1 (simétrica):");
dropperTorch(es_matriz_simetrica(matriz1));

dropperSpider("Matriz 2 (no simétrica):");
dropperTorch(es_matriz_simetrica(matriz2));

dropperSpider("No es matriz:");
dropperSpider(es_matriz_simetrica(noMatriz) >> Spider);

dropperSpider("Matriz rectangular:");
dropperSpider(es_matriz_simetrica(matrizRectangular) >> Spider);
PolloAsado

worldSave

```

9.5.2. Pregunta 2

Pregunta 2. Verificar ganador del *Felis silvestris catus*

Su profesor del semestre entrante los puso a hacer un proyecto programado de un juego de *Felis silvestris catus*. Como ustedes son muy previsores van a dejar listo desde este semestre una función que verifique si ya hay un ganador para el juego.

Ejemplo de matriz de entrada:

```
[[ 'x', 'o', 'x'], ['o', 'x', 'x'], ['-', '-', '-']]
```

Nota: **x** es un jugador, **o** el otro jugador y **-** representa espacios en blanco. Programe una función que permita determinar si hay un ganador en el gato.

La función debe llamarse de la siguiente forma:

```
verificar_ganador_gato(tablero, jugadorx) → True/False
```

donde:

1. **tablero** es una matriz que contiene únicamente textos (sino *'Error01'*)
2. **tablero** es una matriz cuadrada (sino *'Error02'*)
3. **jugadorx** debe ser un texto de un solo caracter (sino *'Error03'*)
4. **jugadorx** debe ser **'x'** o **'o'** únicamente (sino *'Error04'*)
5. La **salida** debe ser un valor de verdad *True* si el jugadorx indicado es ganadorx, *False* en caso contrario.

Figura 9.2: Pregunta 2

Codigo de Notch Engine

WorldName VerificarGanadorGato:

Recipe

```
Spell verificarGato(Shelf Rune :: tablero; Rune :: jugador) -> Torch;
```

CraftingTable

```
Spell verificarGato(Shelf Rune :: tablero; Rune :: jugador) -> Torch
```

PolloCrudo

```
$$ Validaciones básicas
```

```
target #(tablero) isNot 9 craft hit
```

PolloCrudo

```
dropperSpider("Error02"); $$ No es cuadrado (3x3)
```

```
respawn Off;
```

PolloAsado

```
target jugador isNot 'x' and jugador isNot 'o' craft hit
```

PolloCrudo

```

        dropperSpider("Error04"); $$ Jugador no válido
        respawn Off;
PolloAsado

$$ Líneas ganadoras (por índice lineal)
Shelf Shelf Stack lineas = [
    [0,1,2], [3,4,5], [6,7,8],  $$ Filas
    [0,3,6], [1,4,7], [2,5,8],  $$ Columnas
    [0,4,8], [2,4,6]           $$ Diagonales
];

walk i set 0 to 7 craft
PolloCrudo
    Shelf Stack linea = lineas[i];
    Stack a = linea[0];
    Stack b = linea[1];
    Stack c = linea[2];

    target tablero[a] is jugador and tablero[b] is jugador and tablero[c] is
PolloCrudo
    respawn On;
    PolloAsado
PolloAsado

$$ No se encontró ganador
respawn Off;
PolloAsado

SpawnPoint
Shelf Rune gato = ['x','o','x','o','x','x','-','-','-'];
Rune j = 'x';

Torch hayGanador = verificarGato(gato, j);

dropperTorch(hayGanador);

worldSave

```

9.6. Implementación del Generador de Código

9.6.1. Introducción a la Implementación

La implementación del generador de código del compilador Notch Engine se materializa principalmente en el archivo `mc_generacion.py`, que contiene una jerarquía de clases especializadas para diferentes aspectos de la generación de código. Esta implementación modular permite una separación clara de responsabilidades y facilita la extensión y mantenimiento del sistema.

La arquitectura implementada sigue un diseño orientado a objetos que progresa desde funcionalidades básicas hasta características avanzadas, culminando en un generador completo capaz de producir código ensamblador TASM totalmente funcional.

9.6.2. Arquitectura de Clases

Clase Base: `GeneradorCodigo`

La clase `GeneradorCodigo` proporciona la funcionalidad fundamental para la generación de código:

Gestión de Portadas: Genera automáticamente portadas de identificación que incluyen información del proyecto, autores (Cabrera Samir, Urbina Luis), fecha de generación, y metadatos del compilador.

Control de Segmentos: Implementa un sistema robusto de gestión de segmentos de memoria con límites configurables (4KB para datos, 4KB para código) y verificación automática de desbordamiento.

Tabla de Símbolos de Generación: Mantiene una tabla independiente específica para la generación que almacena información como direcciones de memoria, tipos de datos, y metadata específica de cada símbolo.

Plantilla Base: Genera la estructura fundamental del programa ASM incluyendo directivas `ASSUME`, definición de segmentos, y procedimiento principal.

Clase Extendida: `GeneradorConRuntime`

La clase `GeneradorConRuntime` extiende las capacidades básicas incorporando la Runtime Library:

Integración de Runtime Library: Proporciona métodos para incluir automáticamente la Runtime Library en el código generado, ya sea mediante inclusión directa o referencia externa.

Generación de Operaciones: Implementa la traducción de operaciones del lenguaje Notch Engine a llamadas apropiadas de la Runtime Library, manejando la preparación de parámetros y recuperación de resultados.

Creación Automática de Runtime: Incluye funcionalidad para generar automáticamente el archivo completo de Runtime Library con todas las rutinas necesarias.

Clase Completa: `GeneradorCompleto`

La clase `GeneradorCompleto` proporciona funcionalidad avanzada para estructuras de control complejas:

Control de Flujo: Implementa generación de código para estructuras condicionales y repetitivas mediante etiquetas y saltos.

Gestión de Contextos: Mantiene una pila de contextos para manejar estructuras anidadas y garantizar la correcta generación de etiquetas únicas.

Optimizaciones: Incorpora optimizaciones básicas como eliminación de saltos redundantes y reutilización de registros.

9.6.3. Proceso de Generación de Runtime Library

Estructura de la Runtime Library Generada

El método `crear_archivo_runtime` genera un archivo ASM completo que contiene:

Rutinas Aritméticas:

- `SUMAR_ENTEROS`: Implementa operación `:+`
- `RESTAR_ENTEROS`: Implementa operación `:-`
- `MULTIPLICAR_ENTEROS`: Implementa operación `:*`
- `DIVIDIR_ENTEROS`: Implementa operación `:/` con manejo de división por cero
- `MODULO_ENTEROS`: Implementa operación `:%` con protección de errores

Rutinas de Incremento/Decremento:

- `INCREMENTAR_ENTERO`: Implementa `soulsand`
- `DECREMENTAR_ENTERO`: Implementa `netherrack`

Rutinas de Comparación:

- `COMPARAR_IGUAL`: Implementa `is`
- `COMPARAR_DIFERENTE`: Implementa `isNot`
- `COMPARAR_MAYOR`: Implementa `>`
- `COMPARAR_MENOR`: Implementa `<`
- `COMPARAR_MAYOR_IGUAL`: Implementa `>=`
- `COMPARAR_MENOR_IGUAL`: Implementa `<=`

Rutinas Lógicas:

- AND_LOGICO: Implementa operaciones AND
- OR_LOGICO: Implementa operaciones OR
- NOT_LOGICO: Implementa operaciones NOT
- XOR_LOGICO: Implementa operaciones XOR

Rutinas de Entrada/Salida:

- LEER_ENTERO: Lectura de valores STACK
- MOSTRAR_ENTERO: Salida de valores STACK
- LEER_CHARACTER: Lectura de caracteres individuales
- MOSTRAR_CHARACTER: Salida de caracteres
- LEER_BOOLEANO: Lectura de valores TORCH
- MOSTRAR_BOOLEANO: Salida de valores TORCH como ON/OFF

Rutinas de Cadenas:

- CONCATENAR_STRINGS: Unión de cadenas SPIDER
- LONGITUD_STRING: Cálculo de longitud de cadenas

Características de Implementación

Cada rutina de la Runtime Library implementa:

- **Convención de Llamadas Estándar:** Uso de pila para parámetros, retorno en AX, preservación de registros
- **Manejo de Errores:** Detección y manejo seguro de condiciones excepcionales
- **Optimización:** Código eficiente con mínimo uso de recursos
- **Documentación:** Comentarios descriptivos y metadatos de generación

9.6.4. Generación de Código Principal

Proceso de Traducción de Operaciones

El método `generar_operacion_aritmetica` implementa la traducción directa de operaciones Notch Engine:


```
def generar_operacion_aritmetica(self, operador, var_resultado, operando1, operando2):
    operaciones_runtime = {
        '+': 'SUMAR_ENTEROS',
        '-': 'RESTAR_ENTEROS',
        '*': 'MULTIPLICAR_ENTEROS',
        '/': 'DIVIDIR_ENTEROS',
        '%': 'MODULO_ENTEROS'
    }

    codigo_operacion = [
        f"    ; Operación: {var_resultado} = {operando1} {operador} {operando2}"
        f"    PUSH {operando2}    ; Segundo operando",
        f"    PUSH {operando1}    ; Primer operando",
        f"    CALL {rutina}        ; Llamar rutina de Runtime Library",
        f"    MOV {var_resultado}, AX ; Guardar resultado"
    ]
```

Este proceso garantiza que cada operación del lenguaje fuente se traduzca correctamente a una secuencia equivalente de instrucciones ASM que utiliza la Runtime Library.

Declaración de Variables

El método `declarar_variable` maneja la asignación de espacio en el segmento de datos:

- Verifica disponibilidad de espacio en el segmento de datos
- Asigna direcciones de memoria apropiadas según el tipo de dato
- Actualiza la tabla de símbolos de generación
- Genera las directivas ASM correspondientes (DW, DB, etc.)

Control de Flujo

Para estructuras de control, se implementan métodos especializados:

Generación de Etiquetas: Sistema automático de generación de etiquetas únicas para evitar conflictos en programas complejos.

Estructuras Condicionales: Traducción de TARGET/HITMISS a secuencias de comparación y salto condicional.

Bucles: Implementación de estructuras repetitivas mediante etiquetas de inicio y salto condicional de salida.

9.6.5. Archivos de Prueba y Validación

Programa de Prueba (programa_test.asm)

El archivo `programa_test.asm` representa un ejemplo del código generado automáticamente:

```
;=====
;               NOTCH ENGINE COMPILER
;=====
; Generado por: Cabrera Samir, Urbina Luis
; Fecha: 12/06/2025 21:47
; Proyecto: Compilador para Notch Engine
; Etapa 4: Generador de Código
;=====

ASSUME CS:CODE, DS:DATA, SS:STACK

STACK SEGMENT STACK
    DW 256 DUP(?)
STACK ENDS

DATA SEGMENT
    numero DW 10
    ; Operación: resultado = numero :+ 5
    PUSH 5    ; Segundo operando
    PUSH numero ; Primer operando
    CALL SUMAR_ENTEROS ; Llamar rutina de Runtime Library
    MOV resultado, AX ; Guardar resultado
DATA ENDS

CODE SEGMENT
MAIN PROC
    MOV AX, DATA
    MOV DS, AX
    ; Código principal generado
    ; Terminar programa
    MOV AH, 4Ch
    INT 21h
MAIN ENDP
CODE ENDS
END MAIN
```

Este archivo demuestra la estructura completa generada automáticamente, incluyendo portada, segmentos, variables, operaciones y terminación apropiada.

Programa de Prueba de Runtime (`test_runtime_library.asm`)

El archivo `test_runtime_library.asm` contiene pruebas exhaustivas de la Runtime Library:

Pruebas Aritméticas: Verificación de suma, resta, multiplicación, división y módulo con valores conocidos.

Pruebas Lógicas: Validación de operaciones AND, OR, NOT con diferentes combinaciones de valores booleanos.

Pruebas de E/S: Verificación de lectura y escritura para diferentes tipos de datos.

Pruebas de Manejo de Errores: Validación del comportamiento en condiciones excepcionales como división por cero.

Script de Pruebas (`test_generador.py`)

El archivo `test_generador.py` implementa pruebas automatizadas:

```
def test_mc_generacion_directo():
    from mc_generacion import GeneradorConRuntime

    gen = GeneradorConRuntime()

    # Crear Runtime Library
    resultado = gen.crear_archivo_runtime("runtime_generado.asm")

    # Generar código de prueba
    gen.declarar_variable("numero", "STACK", "10")
    gen.generar_operacion_aritmetica(":+", "resultado", "numero", "5")
    gen.finalizar_programa()
    gen.guardar_archivo("programa_test.asm")
```

Este script verifica automáticamente:

- Creación correcta de la Runtime Library
- Generación apropiada de código principal
- Funcionalidad de declaración de variables y operaciones
- Proceso completo de finalización y guardado

9.6.6. Control de Flujo Avanzado

Archivo de Prueba de Control de Flujo (`test_control_flujo_corregido.asm`)

Este archivo demuestra la implementación de estructuras de control complejas:

Estructuras Condicionales Anidadas: Implementación de múltiples niveles de TARGET/HITMISS con etiquetas apropiadas.

Bucles con Condiciones Complejas: Demostración de bucles con múltiples condiciones de salida y variables de control.

Combinación de Estructuras: Integración de diferentes tipos de estructuras de control en un programa cohesivo.

Generación de Etiquetas Únicas

El sistema implementa un algoritmo robusto para generar etiquetas únicas:

- Uso de contadores incrementales para cada tipo de estructura
- Prefijos descriptivos que indican el tipo de estructura (IF_, WHILE_, END_)
- Verificación de unicidad para evitar conflictos en programas complejos
- Mantenimiento de contexto para estructuras anidadas

9.6.7. Integración y Compatibilidad

Runtime Generado (`runtime_generado.asm`)

El archivo `runtime_generado.asm` representa la salida del proceso automático de generación de Runtime Library:

- Contiene todas las rutinas necesarias para la ejecución de programas Notch Engine
- Incluye metadatos de generación y documentación automática
- Implementa manejo robusto de errores y condiciones excepcionales
- Optimizado para eficiencia en tiempo de ejecución

Runtime Library Base (`runtime_library.asm`)

El archivo `runtime_library.asm` sirve como plantilla y referencia:

- Define la estructura estándar para todas las rutinas
- Establece convenciones de codificación y documentación
- Proporciona implementaciones de referencia para operaciones críticas
- Facilita la extensión y mantenimiento de la biblioteca

9.6.8. Manejo de Errores y Robustez

Errores de Generación

El sistema implementa detección y manejo comprehensivo de errores:

Desbordamiento de Segmentos: Verificación automática antes de cada adición de código o datos.

Operadores No Soportados: Validación de operadores antes de generar código correspondiente.

Variables No Declaradas: Verificación de existencia en tabla de símbolos antes de uso.

Tipos Incompatibles: Validación de compatibilidad de tipos antes de generar operaciones.

Estrategia de Recuperación

Cuando se detectan errores durante la generación:

1. Se registra el error con información detallada de contexto
2. Se intenta recuperación automática cuando es posible
3. Se genera código alternativo seguro si la recuperación no es factible
4. Se termina la generación con mensaje descriptivo en casos irre recuperables

9.6.9. Optimizaciones y Eficiencia

Optimizaciones Implementadas

El generador incorpora varias optimizaciones:

Reutilización de Registros: Mantiene valores en registros cuando es beneficioso.

Eliminación de Operaciones Redundantes: Detecta y elimina cálculos innecesarios.

Optimización de Saltos: Minimiza la cantidad de saltos condicionales e incondicionales.

Gestión Eficiente de Pila: Optimiza el uso de la pila para parámetros y variables temporales.

Métricas de Rendimiento

El sistema proporciona estadísticas detalladas:

- Número total de variables declaradas
- Uso actual y disponible de segmentos de datos y código
- Cantidad de operaciones generadas por tipo
- Número de llamadas a Runtime Library
- Eficiencia de uso de memoria

9.6.10. Extensibilidad y Mantenimiento

Arquitectura Modular

La implementación facilita extensiones futuras:

- Separación clara entre generación básica y características avanzadas
- Interfaces bien definidas entre componentes
- Posibilidad de agregar nuevos tipos de datos sin modificar código existente
- Soporte para nuevas operaciones mediante extensión de la Runtime Library

Documentación y Trazabilidad

El código generado incluye:

- Comentarios automáticos que describen cada operación
- Metadatos de generación con fecha, versión y autores
- Trazabilidad entre código fuente y código generado
- Información de depuración para facilitar el mantenimiento

9.6.11. Casos de Uso y Ejemplos Prácticos

Generación de Operaciones Complejas

El sistema maneja casos complejos como:

Expresiones Anidadas: Evaluación de expresiones con múltiples operadores mediante uso apropiado de la pila y llamadas secuenciales a la Runtime Library.

Asignaciones Múltiples: Generación eficiente de código para asignaciones que involucran múltiples variables y operaciones.

Operaciones con Constantes: Optimización especial para operaciones que involucran valores constantes conocidos en tiempo de compilación.

Gestión de Tipos de Datos

El generador maneja apropiadamente todos los tipos del lenguaje Notch Engine:

Tipos Simples: STACK (enteros), GHASt (flotantes), TORCH (booleanos), SPIDER (cadenas).

Tipos Compuestos: SHELF (arreglos) con cálculo automático de direcciones y verificación de límites.

Constantes: OBSIDIAN con verificación de inmutabilidad durante la generación.

Archivos: BOOK con manejo de recursos y verificación de estado.

9.6.12. Integración con el Sistema Completo

Interfaz con Etapas Previas

El generador se integra seamlessly con:

Analizador Léxico: Utiliza información de tokens para generar comentarios descriptivos y mantener trazabilidad.

Analizador Sintáctico: Sigue la estructura sintáctica validada para garantizar correctitud en la generación.

Analizador Semántico: Utiliza información de tipos y alcances de la tabla de símbolos semánticos para generar código apropiado.

Salida Final

El resultado del proceso de generación incluye:

- Archivo principal ASM con el programa compilado
- Archivo de Runtime Library con todas las rutinas necesarias
- Archivos de prueba para validar la funcionalidad
- Documentación automática y metadatos de generación
- Estadísticas de compilación y uso de recursos

La implementación del generador de código representa así una solución robusta, eficiente y extensible que completa exitosamente el proceso de compilación del lenguaje Notch Engine, produciendo código ensamblador TASM de alta calidad que preserva fielmente la semántica del programa fuente mientras proporciona las herramientas necesarias para la ejecución, depuración y mantenimiento del software generado.