



Instituto Tecnológico de Costa Rica

Escuela de Computación

Compiladores e Intérpretes, IC5701

Etapas: Analizador Sintáctico (Parser)

Estudiantes:

Samir Cabrera Tabash, 2022161229

Luis Urbina Salazar, 2023156802

Primer semestre del año 2025

Índice general

Índice general	1
1. Documentacion del Parser	3
1.1. Documentacion inicial	3
1.2. Gramática del Parser	3
1.2.1. Resultados de GikGram	13
1.2.2. Clase Parser	14
1.2.3. Funciones auxiliares	15
1.2.4. Características relevantes	16
1.2.5. SpecialTokens	16
1.2.6. TokenMap	17
1.2.7. GLadosDerechos	17
1.2.8. GNombresTerminales	17
1.2.9. Gramatica	17
1.2.10. Tabla Follows	18
1.2.11. Tabla Parsing	18
1.3. Resultados	18

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

Capítulo 1

Documentacion del Parser

1.1. Documentacion inicial

1.2. Gramática del Parser

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

Ventajas de usar GikGram

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

Errores LL(1) que GikGram puede detectar

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

1. Símbolos terminales o no terminales no definidos.

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

Estructura general de la gramática

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

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

Formato de la gramática

Se usó una notación BNF simplificada, donde:

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

Definición inicial

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

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

Variables & Constantes

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

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

```

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

```

Funciones

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

```

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

```

Statements

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

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

Control

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

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

```

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

```

Asignación

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

```

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

```

Expresiones

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

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

```

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

```

Relaciones

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

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

```

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

```

Aritmética

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

```

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

```

```

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

```

Funciones

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

```

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

```

Acceso

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

```

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

```

```

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

```

Type

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

```

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

```

Literal

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

```

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

```

```

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

```

Elementos

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

```

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

```

1.2.1. Resultados de GikGram

En la Figura 1.1 se muestra un extracto de la validación LL(1) generada por GikGram. Esta validación ayudó a depurar múltiples errores como doble predicción, recursividad por la izquierda y símbolos no alcanzables, logrando así una versión funcional.

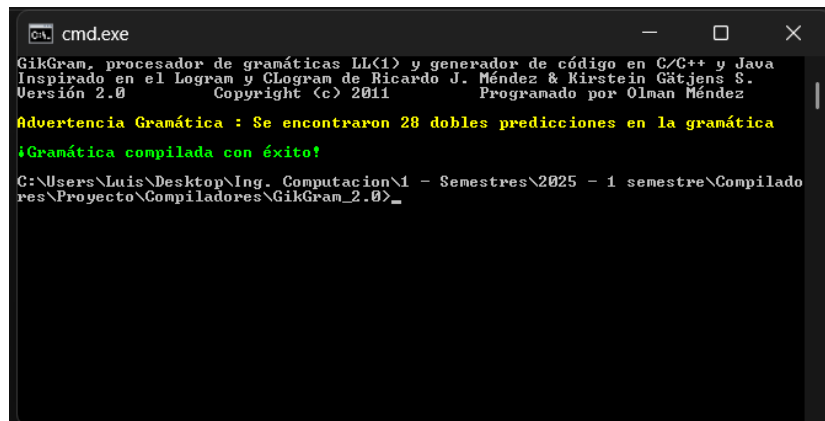


Figura 1.1: Reporte de validación generado por GikGram

Documentación del código

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

1.2.2. Clase Parser

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

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

1.2.3. Funciones auxiliares

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

1.2.4. Características relevantes

El parser implementa varias técnicas avanzadas, como:

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

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

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

1.2.5. SpecialTokens

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

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

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

1.2.6. TokenMap

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

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

1.2.7. GLadosDerechos

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

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

1.2.8. GNombresTerminales

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

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

1.2.9. Gramatica

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

1.2.10. Tabla Follows

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

1.2.11. Tabla Parsing

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

- **MARCA_DERECHA:** Código de familia del terminal de fin de archivo.
- **NO_TERMINAL_INICIAL:** Número del no-terminal inicial.
- **MAX_LADO_DER:** Número máximo de columnas en los lados derechos.
- **MAX_FOLLOWS:** Número máximo de follows.

Métodos principales:

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

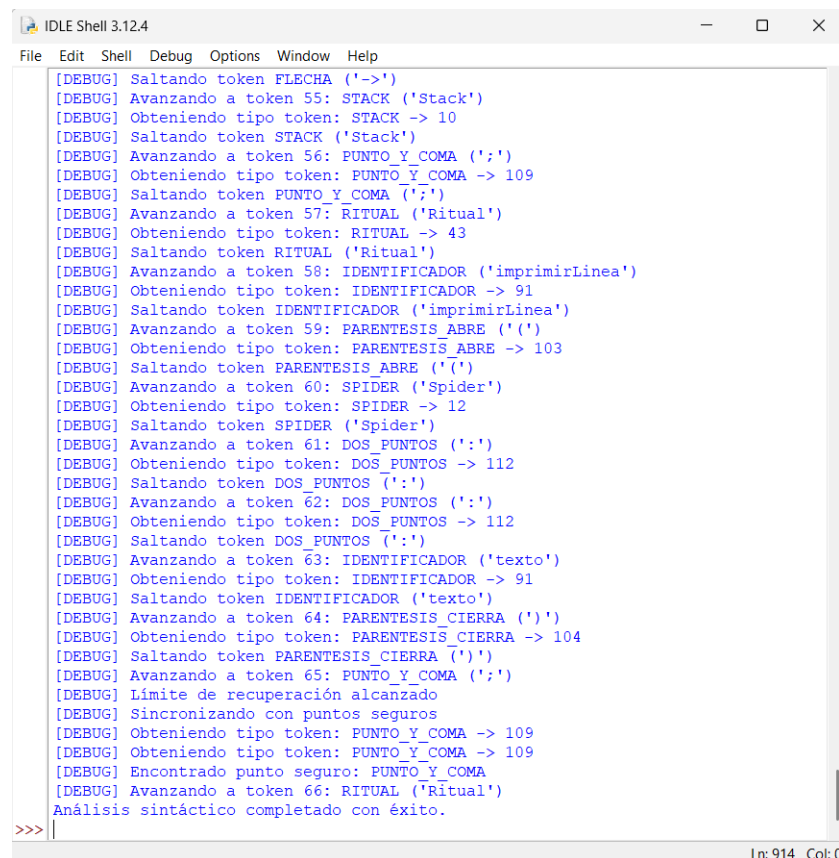
1.3. Resultados

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

Pruebas sin errores

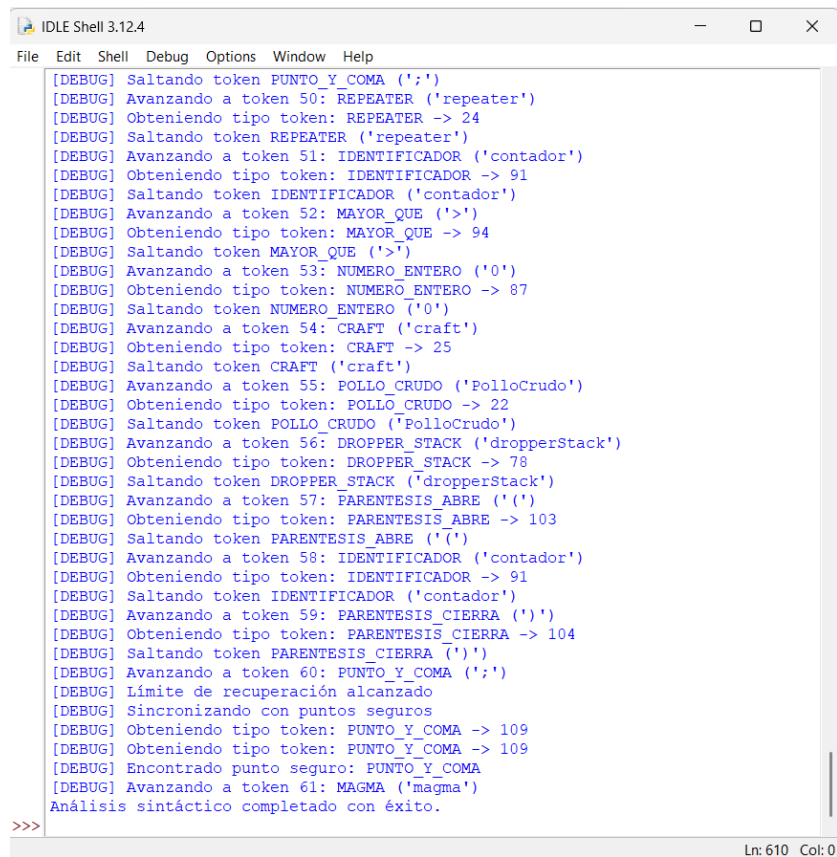
- **07_Prueba_PR_Funciones.txt**: Evalúa el manejo de declaraciones e invocaciones de funciones (**Spell**) y procedimientos (**Ritual**), incluyendo retorno con **respawn**, parámetros múltiples, llamados anidados y estructuras de control dentro del cuerpo de las funciones. Esta prueba pasó sin errores,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

Capturas de pantalla:



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

Figura 1.2: Ejecución exitosa de 07_Prueba_PR_Funciones.txt



```

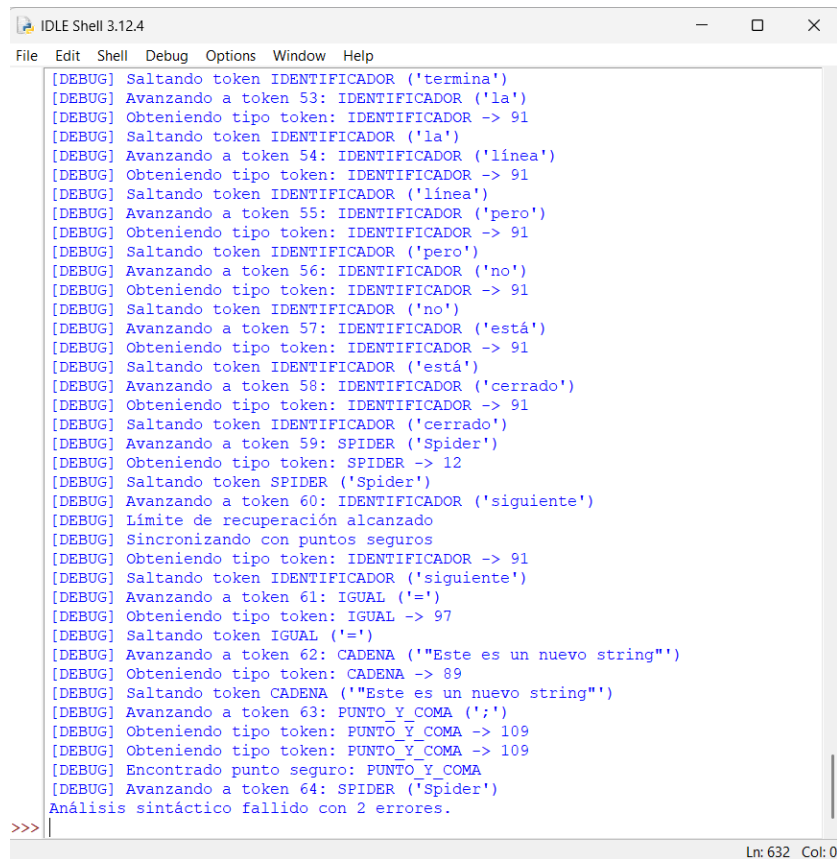
IDLE Shell 3.12.4
File Edit Shell Debug Options Window Help
[DEBUG] Saltando token PUNTO_Y_COMA (';')
[DEBUG] Avanzando a token 50: REPEATER ('repeater')
[DEBUG] Obteniendo tipo token: REPEATER -> 24
[DEBUG] Saltando token REPEATER ('repeater')
[DEBUG] Avanzando a token 51: IDENTIFICADOR ('contador')
[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 1.3: Ejecución exitosa de 05_Prueba_PR_Control.txt

Pruebas con errores detectados

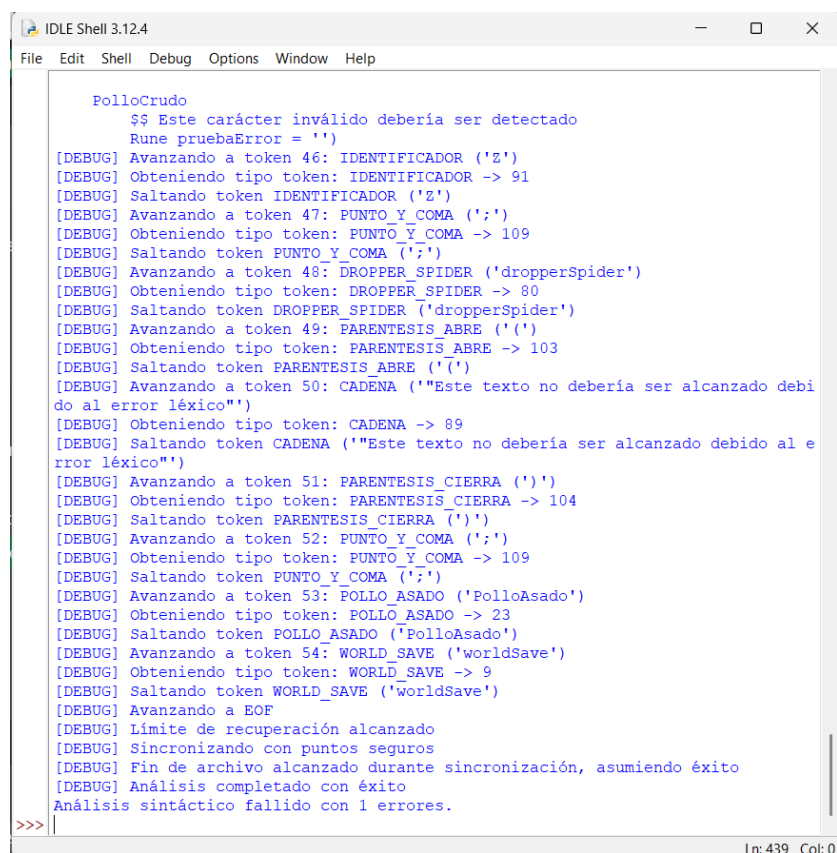
- **35_Prueba_Err_StringNoTerminado.txt**: Contiene múltiples casos de cadenas de texto mal formadas (sin comilla de cierre, seguidas por comentarios o nuevos tokens). El analizador léxico identificó correctamente los errores de forma y generó mensajes adecuados, además de mostrar recuperación parcial 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 1.4: 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 debido al error léxico")
[DEBUG] Obteniendo tipo token: CADENA -> 89
[DEBUG] Saltando token CADENA ("Este texto no debería ser alcanzado debido al error 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 1.5: 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.