



Instituto Tecnológico de Costa Rica

Escuela de Computación

# **Compiladores e Intérpretes, IC5701**

Etapas tres: Analizador Contextual  
(Semántico)

## **Estudiantes:**

Samir Cabrera Tabash, 2022161229

Luis Urbina Salazar, 2023156802

Primer semestre del año 2025

# Índice general

Índice general	1
<b>1. Documentacion del Parser</b>	<b>3</b>
1.1. Documentacion inicial	3
1.2. Gramática del Parser	3
1.2.1. Clase Parser	22
1.2.2. Funciones auxiliares	24
1.2.3. Características relevantes	24
1.2.4. SpecialTokens	24
1.2.5. TokenMap	25
1.2.6. GLadosDerechos	25
1.2.7. GNombresTerminales	25
1.2.8. Gramatica	26
1.2.9. Tabla Follows	26
1.2.10. Tabla Parsing	26
1.3. Resultados	27
<b>2. Documentación del Análisis Semántico</b>	<b>32</b>
2.1. Diccionario Semantico	34
2.1.1. Valor Tipo	34
2.1.2. Chequeo de existencia de identificador	35
2.1.3. Chequeo de constantes	35
2.1.4. Chequeo de Overflow	36
2.1.5. Chequeo de Pollo	37
2.1.6. Chequeo de Shelf Size	37
2.1.7. Chequeo de tipo de operación	38
2.1.8. Chequeo de Uso de Variables	38
2.1.9. Chequeo de WorldName	39
2.1.10. Chequeo de WorldSave	39
2.1.11. Chequeo de Nombre de Archivo	40
2.1.12. Chequeo de Igualdad de Operadores	40

2.1.13.	Chequeo de Inicialización de Variables . . . . .	41
2.1.14.	Chequeo de División por Cero . . . . .	41
2.1.15.	Chequeo de Estructura Target y Hit Miss . . . . .	42
2.2.	Manejo de Tipo de Datos . . . . .	43
2.2.1.	Manejo de Tipos <b>STACK</b> , <b>GHA</b> <b>ST</b> , <b>TORCH</b> , <b>SPIDER</b> y <b>RUNE</b> . . . . .	43
2.2.2.	Manejo de Tipo <b>BOOK</b> . . . . .	44
2.2.3.	Manejo de Tipo <b>CHEST</b> . . . . .	44
2.2.4.	Manejo de Tipo <b>SHELF</b> . . . . .	45
2.2.5.	Manejo de Tipo <b>ENTITY</b> . . . . .	45
2.2.6.	Manejo de Tipo <b>SPELL</b> . . . . .	46
2.2.7.	Manejo de Tipo <b>RITUAL</b> . . . . .	46
2.3.	Explicación del Código Semántico . . . . .	48
2.3.1.	Modificación del Parser . . . . .	48
2.3.2.	Símbolos . . . . .	48
2.3.3.	Verificación Pasada de Tokens . . . . .	49
2.3.4.	Verificación Futura de Tokens . . . . .	49
2.3.5.	Historial Semántico . . . . .	50
2.3.6.	Errores Terminales . . . . .	50
2.4.	Explicación de la Tabla Semántica . . . . .	52
2.4.1.	Estructura de la Tabla Semántica . . . . .	52
2.4.2.	Análisis de Funcionamiento . . . . .	52
2.4.3.	Resultados de la Tabla de Símbolos . . . . .	53
2.4.4.	Ejemplos reales de la tabla semantica . . . . .	54
2.5.	Resultados de Semantica . . . . .	55
2.5.1.	Resultados de la tabla de hash . . . . .	56
2.5.2.	Resultados de historial semantico . . . . .	57
2.5.3.	Resultados de historial semantico negativo . . . . .	59

# Capítulo 1

## Documentacion del Parser

### 1.1. Documentacion inicial

### 1.2. 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<sub>path</sub>'*) *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<sub>e</sub>expr'*) *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;
```

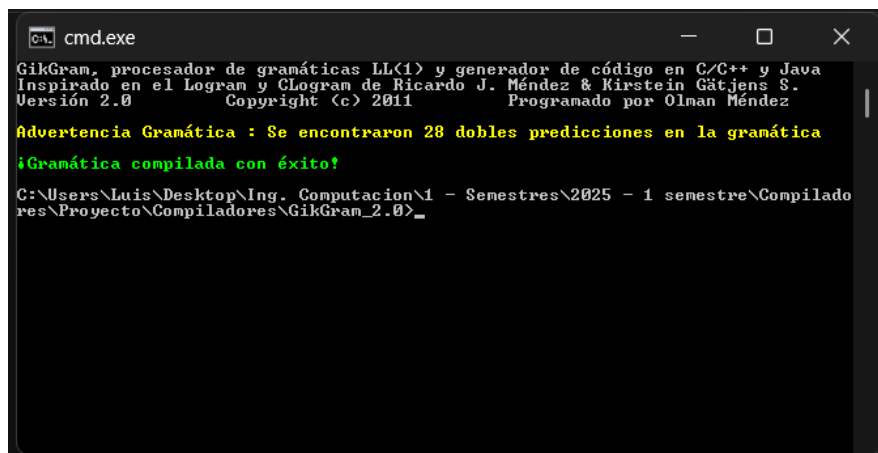
```

    } 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.



```

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 1.1: Reporte de validación generado por GikGram

### Principales errores corregidos con GikGram

- **Rekursividad 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.

### 1.2.1. 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.



### 1.2.2. 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.

### 1.2.3. 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:

### 1.2.4. 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.

### 1.2.5. 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.

### 1.2.6. 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.

### 1.2.7. 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.

### 1.2.8. 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.

### 1.2.9. Tabla Follows

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

### 1.2.10. 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.

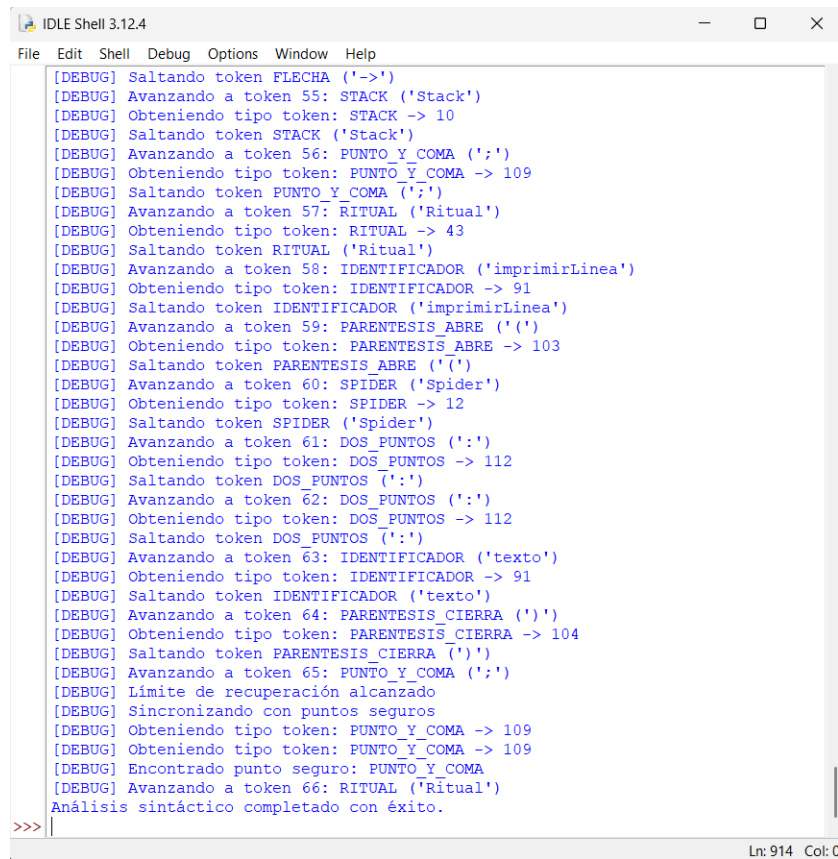
## 1.3. Resultados

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 **wither**. El archivo fue aceptado correctamente, confirmando el soporte completo de instrucciones de control en el parser.

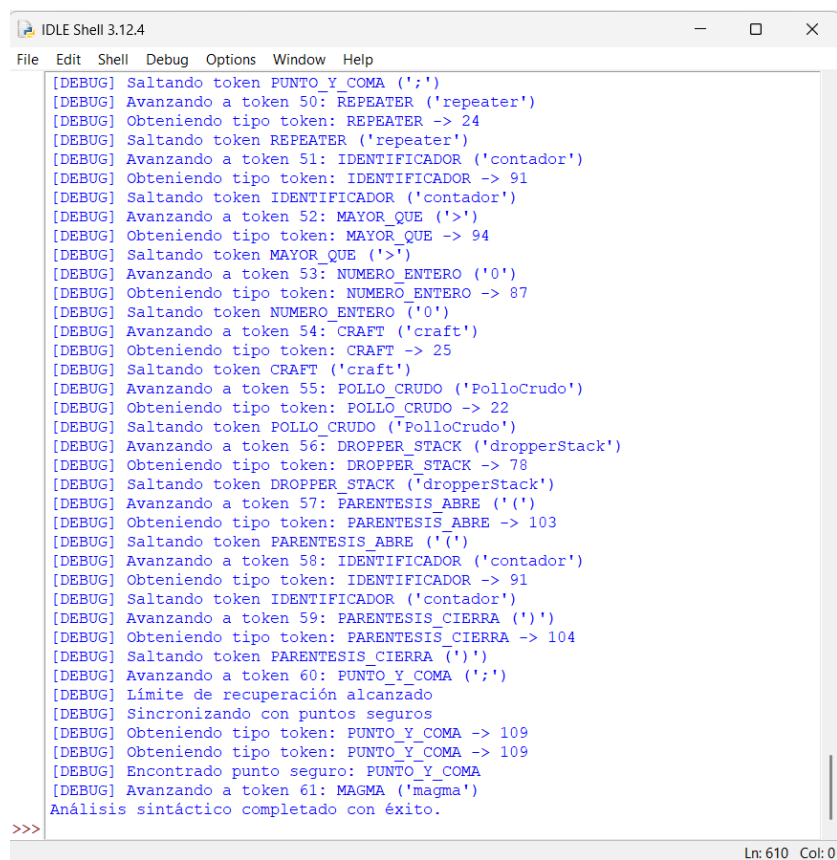
**Capturas de pantalla:**



```
[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 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.
>>>

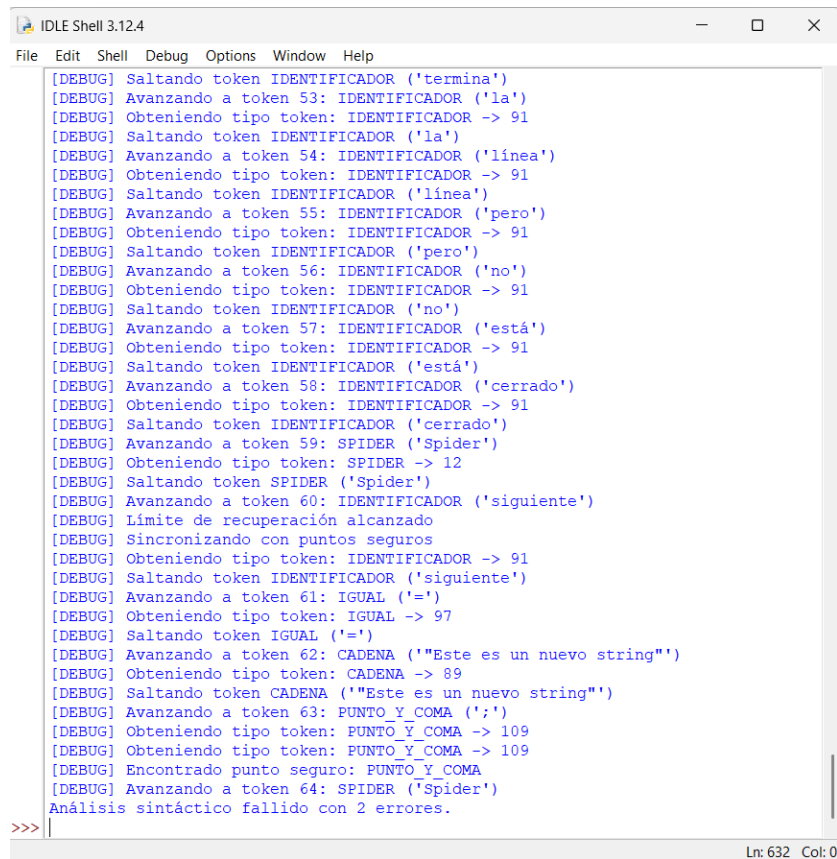
Ln: 610 Col: 0
```

Figura 1.3: Ejecución exitosa de 05\_Prueba\_PR\_Control.txt

## Pruebas con errores detectados

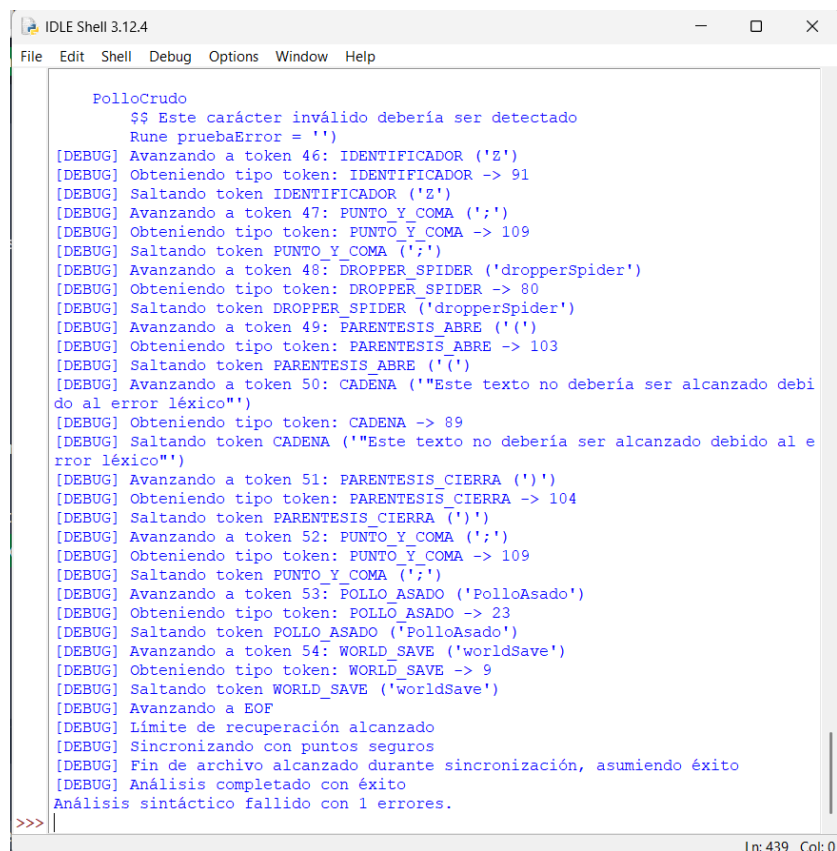
- **35\_Prueba\_Err\_StringNoTerminado.txt**: Contiene múltiples casos de cadenas de texto mal formadas (sin comilla de cierre, seguidas por comentarios o nuevos tokens). El analizador léxico identificó correctamente los errores de forma y generó mensajes adecuados, además de mostrar recuperación parcial 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.



## Capítulo 2

# 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.

## 2.1. Diccionario Semantico

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

### 2.1.1. 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.

### **2.1.2. 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.

### **2.1.3. 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.

## **2.1.4. 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.

### 2.1.5. 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.

### 2.1.6. 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.

## **2.1.7. 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.

## **2.1.8. 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.

## **2.1.9. 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`.

## **2.1.10. 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.

### 2.1.11. 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.

### 2.1.12. 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.

## **2.1.13. 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.

## **2.1.14. 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.

## **2.1.15. 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.

## 2.2. 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:

### 2.2.1. 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.

### 2.2.2. 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.

### 2.2.3. 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**.

#### 2.2.4. 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.

#### 2.2.5. 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.

### 2.2.6. 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.

### 2.2.7. 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.



## 2.3. 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.

### 2.3.1. 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.

### 2.3.2. 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

```

### 2.3.3. 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.

### 2.3.4. 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.

### 2.3.5. 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.

### 2.3.6. 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.

## 2.4. Explicación de la Tabla Semántica

### 2.4.1. 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*.

### 2.4.2. 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.

### 2.4.3. 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 2.1: 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.

#### 2.4.4. 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.

## 2.5. 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 anexas los resultados, se muestra esta prueba por ser larga y robusta.



## 2.5.1. 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 2.1: 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
Escriba AN
```

Figura 2.2: 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 2.3: Tabla 3

## 2.5.2. 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 2.4: 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 2.5: 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 2.6: 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 2.7: 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 2.8: 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': 'string'}'.
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': 'void'}'.
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 000: 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 2.9: Historial Semantico Completo 6

### 2.5.3. 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 'onGate' (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 2.10: Historial Semantico Completo NEGATIVA 1

```
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 'onGate' (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.
21. REGLA SEMANTICA 012: La variable 'onSwitch' (tipo: TORCH) fue declarada pero NO ha sido inicializada. Línea: 97, Columna: 9.
22. REGLA SEMANTICA 012: La variable 'offState' (tipo: TORCH) fue declarada pero NO ha sido inicializada. Línea: 99, Columna: 9.
23. REGLA SEMANTICA 012: La variable 'hopper' (tipo: STACK) fue declarada pero NO ha sido inicializada. Línea: 102, Columna: 9.
24. REGLA SEMANTICA 012: La variable 'hopperMinecart' (tipo: STACK) fue declarada pero NO ha sido inicializada. Línea: 103, Columna: 9.
25. REGLA SEMANTICA 012: La variable 'dropper' (tipo: STACK) fue declarada pero NO ha sido inicializada. Línea: 104, Columna: 9.
26. REGLA SEMANTICA 012: La variable 'dropperFunction' (tipo: STACK) fue declarada pero NO ha sido inicializada. Línea: 105, Columna: 9.
27. REGLA SEMANTICA 001: EL IDENTIFICADOR de nombre 'worldNameGenerator' ya existe
28. REGLA SEMANTICA 001: EL IDENTIFICADOR de nombre 'ender_pearl' ya existe
29. REGLA SEMANTICA 001: EL IDENTIFICADOR de nombre 'craftingTableMaker' ya existe
```

Figura 2.11: Historial Semantico Completo NEGATIVA 2