UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

FACULTAD DE INGENIERÍA

DISEÑO, IMPLEMENTACIÓN Y PRUEBA DE UN GENERADOR DE ANALIZADORES LÉXICOS Y SINTÁCTICOS

TESIS

QUE PARA OBTENER EL TÍTULO DE
INGENIERO EN COMPUTACIÓN

PRESENTAN:
SALVADOR SIERRA AVILES
ISAAC BARRANCO SERRANO
ÁNGEL ALBERTO RIVERA BORJA

DIRECTOR DE TESIS:
ING. ADRIÁN ULISES MERCADO MARTINEZ

MÉXICO, CIUDAD DE MÉXICO, 2016

Índice

Agradecimientos

Quiero agradecer a mi familia, a mis amigos y a mis profesores por ayudarme a ser mejor, tanto personal, como profesionalmente.

Agradezco a la UNAM la oportunidad que me brindo para desarrollarme.

En pocas palabras, gracias a todos los que me apoyaron.

Salvador Sierra Aviles

Le agradezco a Dios por haberme dado salud y un cuerpo sano, por acompañarme y guiarme a lo largo de mi carrera.

A la UNIVERSIDAD NACIONAL AUTONOMA DE MÉXICO por darme la oportunidad de estudiar y tener una carrera profesional.

A mis profesores, ya que ellos me enseñaron a superarme cada día, por su paciencia y motivación y porque cada uno ha aportado su granito de arena en mi formación profesional.

A mis padres Isaac y Teresa por apoyarme en todas las circunstancias, por haberme dado la oportunidad de tener una excelente educación en el transcurso de mi vida, por lo buenos valores que me han inculcado, por el apoyo incondicional que en todo momento me brindaron, por el excelente ejemplo a seguir, pero sobre todo por haberme dado la vida.

A mis hermanos Miguel Ángel y Verónica por llenar mi vida de alegrías cuando más lo necesitaba, por estar conmigo en las buenas y en las malas.

A mi familia, ya que todos y cada uno de ellos han destinado tiempo para enseñarme diversas cosas, por brindarme aportes invaluables que servirán para el resto de mi vida ya que son los cimientos de mi desarrollo tanto profesional como en el ambiento personal.

A mis compañeros y amigos por confiar y creer en mí, por llenar de vivencias cada una de las etapas de mi vida que nunca olvidaré.

Son demasiadas las personas que han formado parte de mi vida a las que quisiera agradecerles su amistad, apoyo incondicional, consejos y compañía en los buenos y malos momentos. Algunas siguen presentes y otras en mis recuerdos y en mi corazón, pero sin importar en donde se

encuentren quiero darles las gracias por todo lo que me han brindado, ya que sin ellas no hubiera llegado a ser la persona que soy hoy, a todos ellos Muchas Gracias.

Isaac Barranco Serrano

En primera instancia, agradezco el gran apoyo que nos brindó el profesor Ulises Mercado quien compartió su tiempo con nosotros para llevar a cabo el proyecto de tesis con dedicación y paciencia. Una persona muy profesional capaz de guiar cualquier tipo de proyecto estudiantil y/o laboral. Agradezco en todo momento el apoyo incondicional de mi madre esperanza Borja Navarro y hermano Erick Rivera Borja quienes me brindaron la ayuda necesaria para conseguir el éxito de mi carrera.

Además agradezco el apoyo de mi futura esposa Anel Romero Gonzales quien siempre estuvo ahí, apoyándome psicológicamente sobre cada tropiezo y caída ocurrido en la carrera de ingeniería en computación.

Muchas gracias en general a la UNAM de brindarme todas las herramientas necesarias para desarrollarme como ingeniero en computación en el ámbito laboral.

Ángel Rivera Borja

Introducción

Las herramientas actuales para los análisis léxico y sintáctico presentan dificultades al implementar un lenguaje, debido a que para poder utilizarlas se necesita aprender el lenguaje propio de la mismas para la definición de las expresiones regulares y gramáticas libres de contexto, además de conocer el lenguaje en el cual se encuentra implementadas, ya que dichos lenguajes se combinan dentro de los archivos de estas herramientas, aumentando la complejidad de los mismos.

Por lo anterior expuesto, se presenta la necesidad de desarrollar una nueva herramienta para la implementación y pruebas de lenguajes libres de contexto utilizados en el campo de la computación.

En este trabajo se mostrarán los conceptos sobre los que se basará esta nueva propuesta, posteriormente una introducción sobre algunos programas que actualmente se utilizan para el análisis léxico y sintáctico en los lenguajes de programación, y para finalizar se abordará el diseño, implementación y pruebas de la nueva herramienta, así como los resultados y las conclusiones que se obtengan.

El resultado esperado de este trabajo es el desarrollo de una herramienta para la implementación y pruebas de analizadores léxicos y sintácticos en los cuales el lenguaje en los que sean descritos sean independiente del lenguaje en que se encuentra implementada la nueva herramienta, además de que el lenguaje que describe la parte léxica y sintáctica sea el mismo, reduciendo así la complejidad para el desarrollo y depuración de un nuevo lenguaje a desarrollar.

Capítulo 1: Fundamentos

Introducción

En este capítulo se presenta los conceptos básicos que sustentan al generador de analizadores léxicos y sintácticos al cual recibe el nombre de VERT (que significa verde en francés), propuesto en este trabajo En el capítulo se abordan los conceptos desde un punto de vista más práctico, pero sin dejar de lado las definiciones formales.

1.1 Expresiones regulares

Una expresión regular representa un patrón de una cadena de caracteres. Una expresión regular básica es un solo caracter del alfabeto Σ, el alfabeto es el conjunto de caracteres que pueden ser usados en una expresión regular dada. Teniendo *b* como cualquier caracter del alfabeto Σ, indicamos que la expresión regular **b** identifica el caracter *b* escribiendo L(**b**) = {*b*}. Existen 2 casos especiales que son:

Para describir la cadena vacía utilizamos el metacaracter ε, siendo la expresión regular L(ε) = {**ε**} como una reconocedora de la cadena vacía.

El otro caso es el símbolo que no reconoce ninguna cadena y cuyo lenguaje es el conjunto vacío, este símbolo es **Φ** y lo escribimos como L(Φ) = {}.

Las expresiones regulares tienen 3 operaciones básicas:

-Disyunción, denotada por el metacaracter **|**

Esta es la elección entre alternativas. En la expresión regular r|s siendo r y s expresiones regulares, reconocerá la unión de los conjuntos de cadenas que reconozcan las expresiones regulares r y s.

-Concatenación, siendo la yuxtaposición de caracteres

Una expresión regular formada por una concatenación de expresiones regulares rs, donde r y s son expresiones regulares, reconocerá el lenguaje generado por el conjunto de cadenas que reconoce r seguido del conjunto de cadenas que reconoce s.

-Repetición o clausula, representada por el caracter *****

La expresión regular r*, donde r es una expresión regular, reconoce la concatenación de cero o más veces el lenguaje reconocido por r.

Un ejemplo de expresión regular es:

ae*i

Esta expresión regular reconoce el lenguaje formado por el caracter **a** seguido de cero o más veces el caracter **e,** y finalizando el caracter **i**.

Desde un punto de vista más práctico, el uso de expresiones regulares también se puede encontrar en Oracle a partir de la versión 10g, con la introducción de funciones llamadas **REGEXP** (Regular Expressions) que permite tener un margen más amplio en el procesamiento intensivo de cadenas de caracteres en la base de datos y en los lenguajes SQL y PL/SQL.

A continuación se muestran algunos de los metacaracteres que utiliza Oracle para hacer búsquedas dentro de la base de datos:

- ^ coincide el patrón de búsqueda al inicio de una línea.
- $ coincide el patrón de búsqueda al final de una línea.
- . coincide cualquier caracter en cualquier lugar.
- [] especifica un rango de caracteres

- ? ubica un caracter opcional.
- + ubica uno o más caracteres.
- - ubica cero o más caracteres.
- {n} ubica un caracter que aparece n veces.
- {n,} ubica un caracter que aparece n o más veces.
- {n,m} ubica un caracter que aparece de n a m veces.
- | disyunción o sea un or lógico entre caracteres.

En Oracle las expresiones regulares se utilizan para búsquedas que son demasiado complicadas cuando los comandos SELECT, LIKE, =, son insuficientes para encontrar el resultado esperado. Algunas funciones con las que Oracle cuenta para dichas búsquedas son las siguientes:

REGEXP_LIKE: Función que permite regresar aquellas filas que coinciden con el patrón especificado en una expresión regular.

REGEXP_COUNT: Función que permite contar el número de veces que un patrón aparece en una cadena de caracteres

Un ejemplo sencillo de la aplicación de estas funciones es el siguiente:

Supongamos que tenemos una tabla con el nombre y apellido paterno de los alumnos de una clase.

Esta tabla tendrá como nombre **ALUMNOS**. En esta tabla vienen los siguientes registros:

ID	NOMBRE	APELLIDO_PATERNO
1	MIGUEL	MARTINEZ
2	ESTELA	MORAN
3	BLANCA	REYES
4	ARTURO	DOMINGUEZ
5	ROBERTO	QUEZADA

Tabla 1.1: Tabla de ALUMNOS

Queremos obtener el NOMBRE Y APELLIDO PATERNO de los registros en los cuales el apellido paterno empiecen con la letra M.

Con la ayuda de la función **REGEXP_LIKE** la consulta quedaría de la siguiente manera:

```
SELECT  NOMBRE, APELLIDO_PATERNO FROM ALUMNOS
WHERE REGEXP_LIKE (APELLIDO_PATERNO, '^M');
```

Y el resultado que nos daría sería el siguiente:

NOMBRE	APELLIDO_PATERNO
MIGUEL	MARTINEZ
ESTELA	MORAN

Tabla 1.2: Resultado de consulta de tabla alumnos

El manejo de las expresiones regulares no sólo lo podemos encontrar en Oracle y UNIX, también las podemos utilizar en la mayoría de los lenguajes de programación como lo son: PEARL, JAVA, .NET, PHP entre otros, pero cada lenguaje ocupa su propia sintaxis.

1.2 Gramáticas libres de contexto

Una gramática libre de contexto (GLC) es un conjunto de reglas de producción usadas para reconocer o generar patrones de cadenas. Las GLC están formadas por los siguientes componentes:

-Un conjunto de símbolos terminales, que son los caracteres contenidos en el alfabeto.

-Un conjunto de símbolos no terminales, los cuales contienen los patrones de símbolos terminales y no terminales que generan.

-Un conjunto de producciones, las cuales son reglas para reemplazar símbolos no terminales por símbolos terminales o no terminales.

-Un símbolo inicial que es un símbolo no terminal desde el cual se inicia el reconocimiento o generación de una cadena por parte de la gramática.

La notación Backus-Naur form (BNF):

Es la forma más popular de definir las gramáticas libres de contexto en el campo de la computación, la notación es la siguiente:

-Los no terminales aparecen entre <>

-Una disyunción separa a sus elementos por |

-Un terminal está entre los símbolos “”

-Los símbolos [] indican que lo contenido por ellos puede o no estar

-Una producción está dada por: <nombre_de_produccion>::=contenido de producción

-Los símbolos {} indican que lo que contienen puede repetirse cero o más veces.

Un ejemplo de una gramática libre de contexto en notación BNF es la siguiente:

<expresión> ::= <suma> | <resta> | <multiplicación> | <división> | <números>

<suma> ::= <expresión> “+” <expresión>

<resta> ::= <expresión> “-” <expresión>

<multiplicación> ::= <expresión> “*” <expresión>

<división> ::= <expresión> “/” <expresión>

<número> ::= “0” | “1” | “2” | “3” | “4” | “5” | “6” | “7” | “8” | “9”

<números> ::= <número> [<números>]

1.3 Métodos de análisis sintáctico

1.3.1 Algoritmo LR(k)

Un analizador del tipo LR (Left to right, Rightmost derivation) es de tipo ascendente debido a que a partir de los tokens, trata de deducir cual es la producción apropiada.

Este tipo de analizadores utilizan tablas que dirigen el análisis. Estas tablas son difíciles de hacer a mano, por lo que generalmente son producidas de manera automática por algún generador de analizadores sintácticos.

Una desventaja de este tipo de analizadores es que a medida que k aumenta para poder identificar la gramática, el tamaño de las tablas crece rápidamente. La mayoría de lenguajes de programación pueden ser identificados con analizadores LR con una k igual a uno, por lo que las tablas generadas no son excesivamente grandes.

Las partes que integran un analizador LR son las que se muestran en el diagrama 1.1.

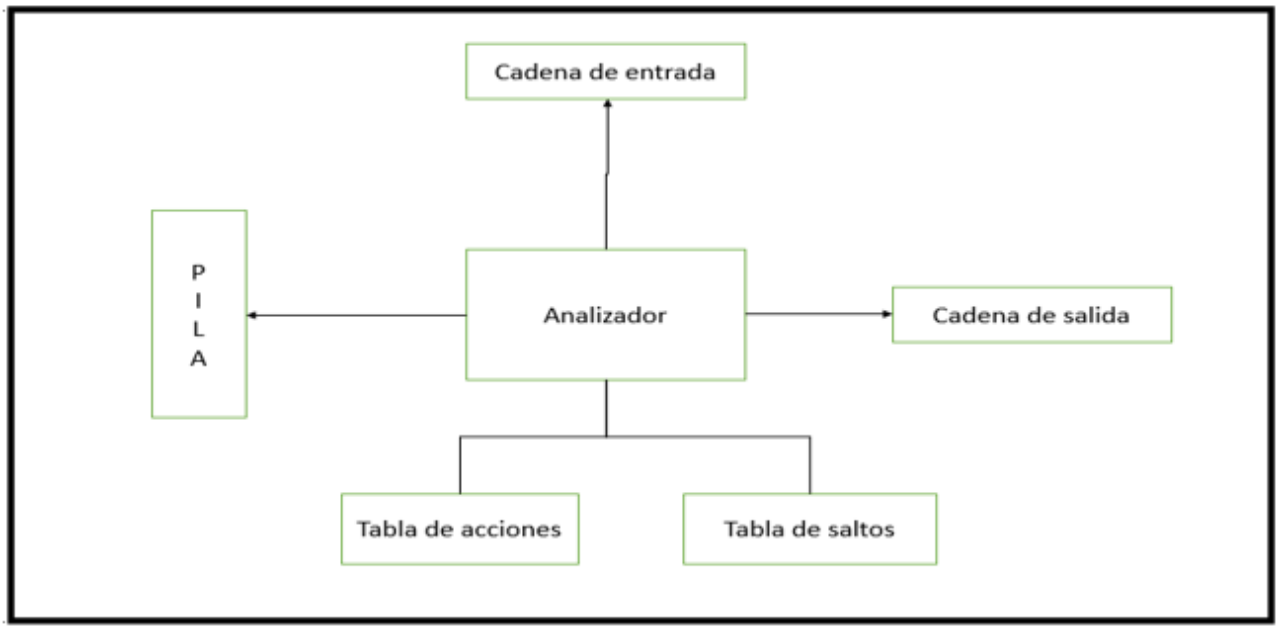


Diagrama 1.1: Estructura del analizador LR

El algoritmo que sigue este tipo de analizador es:

```
poner estado 0 sobre la pila
ciclo
        s = estado en la cima de la pila
        c = siguiente símbolo de entrada
        si ACCION[s, c] = “desplazamiento N” entonces
                poner c sobre la pila
                avanzar en la entrada
                poner estado N sobre la pila
        en otro caso si ACCION[s, c] = “Reducción R” entonces
                dejar regla R ser A ->β
                botar 2 * |β| elementos de la pila
                s’ = estado actual en la cima de la pila
                poner A en la cima de la pila
                poner IRA[s’, A] en la cima de la pila
                imprimir “A->β”
        en otro caso si ACCION[s, c] = “aceptar” entonces
                devuelve éxito
        en otro caso
                devuelve “error de sintaxis”
        fin de si
fin de ciclo
```

Para la gramática:

A → (A) | a

Su tabla respectiva tabla es:

Estado	Acción	Regla	Entrada			Ir a
			(	a	)	A
0	desplazamiento		3	2		1
1	reducción	A'→A				
2	reducción	A→ a				
3	desplazamiento		3	2		4
4	desplazamiento				5	
5	reducción	A→(A)				

Tabla 1.3: Ejemplo de tabla de saltos

Para la entrada:

((a))

Las acciones para el análisis sintáctico serían:

	Pila	Entrada	Acción
1	$0	((a))$	desplazamiento
2	$0(3	(a))$	desplazamiento
3	$0(3(3	a))$	desplazamiento
4	$0(3(3a2	))$	reducción A → a
5	$0(3(3A4	))$	desplazamiento
6	$0(3(3A4)5	)$	reducción A → (A)
7	$0(3A4	)$	desplazamiento
8	$0(3A4)5	$	reducción A → (A)
9	$0A1	$	aceptación

Tabla 1.4: Ejemplo de tabla de acciones de analizador LR

1.3.1.1 Algoritmo LALR(1)

Los analizadores LALR (Look-Ahead LR Parser) son una simplificación de sus pares LR, teniendo la ventaja de generar menos estados que un analizador LR para tener una eficiencia en memoria mayor, pero teniendo el problema de no poder reconocer todos los lenguajes que sí puede un analizador LR.

La simplificación, que realizan los analizadores LALR, es unir las producciones que tienen idénticos núcleos de conjunto de elementos, ya que los elementos siguientes, necesarios para resolver la ambigüedad, no son conocidos en la fase de construcción de LR(0) de construcción de estados. Este tipo de simplificación puede llevar a conflictos del tipo Reducción/Reducción.

Gran variedad de lenguajes de programación actuales pueden ser procesados de manera correcta utilizando analizadores LALR(1), que aunque menos poderosos que los LR(1), tienen una gran capacidad de reconocimiento.

1.3.2 Algoritmo LL(k)

Los analizadores sintácticos que utilizan este tipo de algoritmo son del tipo descendente en los cuales el procesamiento comienza con la producción raíz y desciende a las producciones hasta llegar a los nodos terminales del árbol sintáctico y, de esta manera, ir consumiendo la cadena de entrada.

El algoritmo de este tipo de analizadores descendentes es:

-Inicia la ejecución en la producción inicial

-Las producciones van llamando a otras producciones o verificando no terminales con el caracter actual según sea el caso.

-En caso de completarse correctamente la producción inicial, la cadena ha sido aceptada.

Cabe notar que los algoritmos de este tipo de analizadores también pueden ser efectuados mediante tablas como en los algoritmos anteriores.

Las gramáticas de este tipo pueden ser procesadas por analizadores descendientes recursivos, los cuales tienen una estructura idéntica a la gramática que reconocen. Estos utilizan una pila que lleva el orden de las instancias de las producciones que se están ejecutando y en qué puntos de ejecución se encuentran, permitiendo la recursividad.

Este tipo de algoritmos tienen que ser implementados con gramáticas que no sean recursivas por la

izquierda, esto es que un símbolo no terminal, en una producción, conduzca a la misma producción de manera directa o indirecta(A -> Aα|β), provocando una recursividad indefinida, es decir, que no termina.

La creación de analizadores recursivos descendentes suele ser hecha a mano, lo cual puede ser laborioso y poco viable para gramáticas de cierta complejidad como las que son utilizadas para lenguajes de programación reales.

La variable k representa el número de tokens que el analizador necesita ver hacia adelante "lookahead" para poder resolver que producción de la gramática es la correcta para un punto de la cadena de entrada, para resolver este tipo de casos los analizadores recursivos descendente utilizan "backtracking" que es la prueba de las diferentes producciones posibles, una por una, que comparta uno o varios tokens como prefijos hasta encontrar la producción correcta. Este tipo de procesamiento tiene como inconveniente que su complejidad puede llegar a ser exponencial.

Para la gramática:

$$A \rightarrow (A) \mid a$$

Para la misma entrada que el ejemplo anterior:

((a))

El proceso con un analizador recursivo descendente:

	Pila	Entrada	Acción
1	$A	((a))$	Inicia una nueva instancia de A
2	$A	(a))$	Consume (
3	$AA	(a))$	Inicia una nueva instancia de A
4	$AA	a))$	Consume (
5	$AAA	a))$	Inicia una nueva instancia de A
6	$AAA	))$	Consume a
7	$AA	))$	Se cierra última instancia de A
8	$AA	)$	Consume)
9	$A	)$	Se cierra última instancia de A
10	$A	$	Consume)

Tabla 1.5: Proceso de ejemplo de analizador LL

Capítulo 2: Análisis de herramientas actuales para la generación de compiladores

Introducción

En este capítulo se mostrarán y analizarán algunas herramientas populares para el análisis léxico y sintáctico para el diseño de un lenguaje libre de contexto entre los que se hallan algunos lenguajes de programación. Se analizarán algunas características de estas herramientas y con base en este análisis se justificará la creación de una nueva herramienta.

2.1 Lex

Es un programa de computadora que genera analizadores Léxicos. Lex lee una cadena de entrada y dada una especificación de un analizador léxico por medio de expresiones regulares y código C a ejecutar, realiza acciones al encontrar coincidencias de la cadena de entrada y los patrones declarados.

Usualmente Lex es utilizado, por su velocidad, por generadores de analizadores sintácticos para la etapa de análisis léxico. Los generadores de analizadores sintácticos hacen uso de gramáticas libres de contexto para llevar a cabo un análisis que no se puede realizar con expresiones regulares.

Generalmente Lex es usado con generadores de analizadores sintácticos. Los generadores sintácticos utilizan gramáticas libres de contexto para llevar a cabo un análisis que no se puede hacer con expresiones regulares.

La estructura de un archivo de Lex

Un archivo Lex está dividido en tres partes, las cuales son las siguientes:

Sección de definición
%%
Sección de reglas
%%
Sección código C

Tabla 2.1 Partes de un archivo de lex

La **sección de definición** contiene macros y archivos de cabeceras en C a importar. El código C escrito en esta sección se copiará al archivo generado. Esta sección contiene:

-Declaraciones para el manejo de tablas de Lex

-Las condiciones iniciales

-La **sección de reglas** asocia expresiones regulares con acciones en código C. Las acciones son disparadas cuando un patrón de una expresión regular es encontrado en la cadena de entrada y si un segmento de la cadena de entrada no es reconocido por alguna expresión regular declarada, es mostrada en la salida estándar. Esta sección contiene:

- Formato de asociación de patrones y acciones: expresión regular { acciones }

- Código C que será copiado textualmente al archivo generado

- La **sección de código C** contiene código C que será copiado textualmente al archivo de código fuente asociado, además de que en esta sección se realiza el manejo de la tabla de símbolos. El código C es llamado por las reglas declaradas en la sección de reglas.

Enseguida se muestra un ejemplo de un archivo de Lex

```
/*** Sección de definición ***/

/* Código C que será copiado */

%{

#include <stdio.h>

%}

%option noyywrap /* Indicamos que la entrada será solo un archivo */

%%

/*** Sección de reglas ***/

/* la cadena de coincidencia encontrada con la expresión regular asociada, es la variable

yytext */

[a-z]+  {printf("Identificador %s\n", yytext);}

[0-9]+  {printf("Numero %s\n", yytext);}

.|\n    {   /* Al no realizar ninguna acción para el resto de caracteres, los ignorará */   }

%%

/*** Sección de código C ***/

int main(void)

{

/* llamamos al analizador léxico y terminamos el programa */

yyLex(); //yylex();

return 0;

}
```

Con las siguiente entrada:

```
32434genial5&/(4543578hola.-&&$%%
```

El programa devolvería:

Numero 32434

Identificador genial

Numero 5

Numero 4543578

Identificador hola

Análisis de Lex

Como podemos observar, el código de lex combina el metalenguaje de Lex y Código C, lo cual aumenta la complejidad de pasar del diseño conceptual de los patrones de los tokens a la implementación del analizador léxico que implemente estos patrones

2.2 Yacc

Podemos definir a Yacc como un programa o compilador para generar analizadores sintácticos del tipo LALR, el cual depende de un analizador léxico como entrada produciendo una salida de tipo sintáctica. El analizador sintáctico generado verifica que una secuencia de tokens sea válida para una gramática libre de contexto. Yacc genera una función llamada yyparse donde se guarda el analizador sintáctico realizando el análisis gramatical donde, si regresa un valor de cero la gramática analizada es correcta en caso contrario regresa 1 y se clasifica como incorrecta.

Podemos ilustrarlo de la siguiente manera:

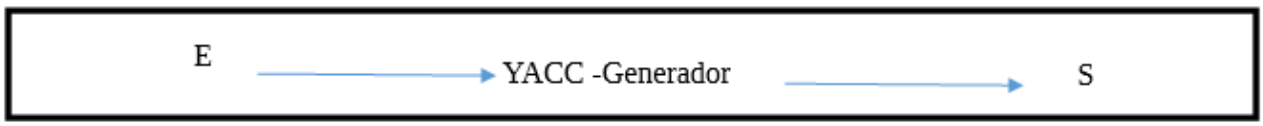


Diagrama 2.1: Creación de analizador sintáctico por Yacc

- Donde **E** es el archivo que contiene la gramática.
- Yacc es un Generador
- **S** : es la salida de Yacc obteniendo un analizador sintáctico

Yacc nos ofrece tres principales características para poder generar analizadores sintácticos

Las declaraciones, reglas de traducción, y rutinas

%%**declaraciones**
%%**reglas**
%%**rutinas en C de apoyo**

Tabla 2.2: Partes de un archivo de Yacc

El separador de secciones se pude mostrar de la siguiente manera %%

Ejemplo: %(a)% marca el inicio y final de cada regla, cada sección se puede definir como una regla.

Sección de declaraciones

Existen tres tipos de funciones que nos ayudarán a manejar Yacc

Una de sus principales características es definir símbolos de la gramática mediante una función llamada "token" definida como sección de declaración en donde se le pasa como argumento símbolos terminales con los que va identificando los posibles valores que van a servir de apoyo funcional para construir el analizador sintáctico. Están funciones actúan como un tipo de bibliotecas.

Sección de reglas

Podemos definir una gramática, en primera instancia encontramos que podemos asignar valores o símbolos no terminales, mientras que en el lado derecho podemos asignar símbolos terminales y no terminales acompañados de una acción a ejecutar escrita en C.

Ejemplo

< Lado izquierdo > ->< alt1 > | < alt2 > ... | < altn >

<Lado izquierdo> :< alt1 > {acción semántica 1} |

< alt2 > {acción semántica 2} ... |

< altn > {acción semántica n} ;

Sección de rutina

En esta parte se debe incluir una función llamada yylex(). La cual se encarga de producir pares formados por un componente léxico y su valor, es decir los tokens. Si se devuelve un componente léxico como HOLA, el componente léxico se debe declarar en la primera sección de la especificación en Yacc.

Se deben declarar variables dentro de un archivo para que pueda reconocer las distintas funciones del programa, por ejemplo main ().

Análisis de Yacc

Yacc es un analizador sintáctico que depende de un analizador léxico, por ejemplo Lex. Su dependencia reside en que los tokens son generados por Lex y estos son utilizados por Yacc para el análisis sintáctico. Con lo anterior se puede ver que la complejidad del diseño e implementación de los análisis léxico y sintáctico aumenta, ya que no solo se tiene que conocer los metalenguajes de Yac y Lex, sino que también se tiene que conocer el lenguaje C, debido a que es implementado en los archivos tanto en Lex como en Yacc.

2.3 Javacc

Javacc (Java Compiler Compiler) es un metacompilador (un compilador de compiladores, un programa que acepta como parámetro de entrada una gramática y en la salida genera un autómata que describe ese lenguaje) que construye un programa que es capaz de reconocer elementos de un lenguaje especifico con una salida en código java, a diferencia de otros metacompiladores como lex/yacc que en estos la salida se genera en código C.

Los analizadores sintácticos que son generados por Javacc son analizadores sintácticos descendentes (intenta encontrar entre las producciones de la gramática la derivación por la izquierda del símbolo inicial de la cadena que se le manda como parámetro de entrada).

Las definiciones léxicas pueden estar incluidas dentro de lo que se especifica en la gramática por ejemplo para generar un token para el comando “IF” y para “(“ se puede generar de la siguiente manera:

SENTENCIAIF : {} { “if” “(“ expresión() “)” sentencia () }

Y con esto generamos tokens para “if” y para “(“. También dentro de este metacompilador además del token existen conceptos como more y skip entre otros, lo cual hace un manejo más fácil y claro en el uso de especificaciones y permite una mejor gestión en el manejo de excepciones y mensajes de error lo cual contiene información muy precisa respecto al origen del error y su posición.

Otra característica de Javacc es que las especificaciones léxicas y sintácticas de la gramática que se quiere analizar se incluyen en un mismo archivo.

En este metacompilador se incluyen dos utilerías:

JJTree: Sirve para generar arboles sintácticos

JJDoc: Convierte archivos de la gramática en archivos para documentación.

A continuación se muestra un ejemplo breve de cómo funciona Javacc:

Javacc para identificar los archivos de entrada que contienen las especificaciones de la gramática que se va a analizar utiliza la extensión .jj, lo cual se refiere que es un archivo que puede ser compilado por Javacc, el siguiente es un ejemplo de un archivo de Javacc:

```
PARSER_BEGIN (tesis)  -----Nombre de la clase (tesis es el nombre del archivo tesis.jj que va
a  contener las especificaciones de la gramatica)
class tesis
{
public static void main (String [] args) throws ParseException  ---Esto permitirá manejar las
excepciones que genere el
compilador.
    {
try
          {
tesis analizador=new tesis(System.in); -----instancia de nuestra clase
                analizador.Programa_tesis(); ---Método principal de nuestra sintaxis
          }

catch(ParseException e) ------Aquí vamos a manejar la excepción ParseException
          {
System.out.println(e.getMessage()); ----Imprimimos en la consola el mensaje que nos
manda la excepción
System.out.println("Termino con error"); ---Imprimimos mensaje para error
          }
    }
}
PARSER_END(tesis)  -----Cerramos el PARSE
------BLOQUE DE TOKEN´S
TOKEN: ----------Token para palabras reservadas
```

```
{
<MAIN: "public static void Main() ">{System.out.println("MAIN ->  "+image);}
   |  <PROGRAMA: "Programa" >{ System.out.println("PROGRAMA ->  "+image);}
   |  <IF: "ien" >{ System.out.println("IF -> "+image);}
}
TOKEN: -------Token para separadores
{
<LPAREN: "(" >{System.out.println("IPAREN ->  "+image);}
   | <RPAREN: ")">{System.out.println("DPAREN ->  "+image);}
   | <LBRACE: "{" >{System.out.println("ILLAVE  ->  "+image);}
   | <RBRACE:"}" >{System.out.println("DLLAVE  ->  "+image);}
   | <RPAREN: "(" >{System.out.println("DPAREN ->  "+image);}
| <SEMICOLON: ";">{System.out.println("PUNTO Y COMA ->  "+image);}
}
TOKEN:  ----Token para operadores relacionales
{
<ASIGNACION: "=" >{System.out.println("ASIGNACION ->  "+image);}
   | <MENOR: "<" >{System.out.println("MENOR QUE  ->  "+image);}
   | <MAYOR: ">" >{System.out.println("MAYOR QUE  ->  "+image);}
}
TOKEN: ----Token para operadores aritméticos
{
<NUMBER: (["0"-"9"])+>{System.out.println("NUMERO ->  "+image);}
| <INDENTIFIER: ["a"-"z","A"-"Z"](["a"-"z","A"-"Z","0"-"9","_"]) >{System.out.println("
      IDENTIFICADOR  ->  "+image);}
| <INT: "inum" >{System.out.println("ENTERO  ->  "+image);}
```

}

SKIP: ---------Omite los espacios, saltos de línea y tabulaciones

{

“ ”| “ \r\n” | “\t”

}

----------Empezamos con la sintaxis para la lectura de la gramática

void Programa_tesis () : ---Método principal

{}

{

<PROGRAMA><IDENTIFIER><LBRACE>Inicio () <RBRACE><EOF>

}

void Inicio ():

{}

{

<MAIN>

<LBRACE>Sentencias () <RBRACE>

}

void Sentencias ():

{}

{

DecLocal ()

| SentenciaIf ()

}*----- Esta estrella significa que el método Sentencias se puede repetir cero o mas veces

void DecLocal () :

{}

{

<INT><IDENTIFIER><SEMICOLON> VS ()

}

void VS (): ----Metodo auxiliar

{}

{

DecLocal () | Sentencias ()

}

void SentenciaIf () :

{}

{

<IF><LPAREN> Comparaciones () <RPAREN><LPAREN><RBRACE>

}

void Comparaciones () :

{}

{

Valores () Operadores () Valores ()

}

void Valores () :

{}

{

<IDENTIFIER> | <NUMBER>

}

void Operadores () :

{}

{

<MENOR>|<MAYOR>

```
}
```

------Se arma el siguiente código para empezar con el análisis

```
Programa  tesis2

{

public static void Main ()

{

inum   num4;

ien  (num4   > 6)

        {

       }
   }
}
```

---------Ejecutando el programa la salida del análisis quedaría de la siguiente manera:

PROGRAMA -> Programa

IDENTIFICADOR -> tesis2

ILLAVE -> {

MAIN -> public static void Main ()

ILLAVE -> {

ENTERO -> inum

IDENTFICADOR -> num4

PUNTO Y COMA -> ;

IF -> ien

IPAREN ->(

IDENTFICADOR -> num4

MAYOR QUE ->>

NUMERO -> 6

DPAREN ->)

ILLAVE -> {

DLLAVE -> }

DLLAVE -> }

DLLAVE -> }

Análisis de Javacc

El hecho de que Javacc combine la parte léxica y sintáctica en un mismo archivo, además de que se tenga que conocer el lenguaje de programación Java, hacen no sea trivial la implementación de los análisis léxico y sintáctico.

Capítulo 3: Diseño de una nueva herramienta

Introducción

El presente capítulo abordará el diseño de los diferentes componentes de VERT. Los componentes cuentan con una descripción de su comportamiento, entradas, salidas, integración con los otros componentes y, en algunos casos, optimización.

3.1. Diseño del lenguaje para declaración de las expresiones regulares y gramáticas

Producción inicial

Es el operador desde donde parte el análisis ya sea léxico o sintáctico.

Sintaxis:

<producción_inicial>::= “~” <identificador> “=”

<concatenación_de_operadores>

“!”

El ejemplo siguiente identifica los bloques “if” y “while” declarados en dichas producciones (las producciones son referenciadas con el símbolo “%” antes de su nombre).

~inicio =

(

%if

|

%while

)

¡

Producción normal

Este operador es el encargado de "encapsular" un patrón sintáctico para poder ser referenciado por otros operadores.

Sintaxis:

<producción_normal>::=<identificador> "="

<concatenación_de_operadores>

"!"

En el siguiente ejemplo se muestra la producción llamada "if" que define el patrón de la estructura de control "if" de cierto lenguaje.

if =

if %comparación then

%contenido_de_if

end

!

Rango de repeticiones

Identifica que el operador que le precede se cumpla en un número de veces que este entre el rango definido.

Sintaxis:

<Rango_de_repeticiones>::=<operador> "+" <numero_entero_positivo>

[("," [<numero_entero_positivo>])]

Los ejemplos muestran el número de veces que el operador "Rango de repeticiones" aceptará a la cadena "Hola":

De 1 a 10 veces:

Hola +1,10

Cero o más veces:

Hola +0,

Por lo menos una vez:

Hola +1,

Exactamente 7 veces:

Hola +7

Disyunción sintáctica

Este operador define varias alternativas a identificar.

Sintaxis:

<Disyunción sintactica>::= "(" <concatenación_de_operadores>

"|" <concatenación_de_operadores>

{ "|" <concatenación_de_operadores> }

")"

Este ejemplo identifica la cadena "genial" o la producción "suma":

(genial | %suma)

Concatenación sintáctica

Como su nombre lo indica, concatena 2 o más operadores para que identifiquen de manera secuencial.

Sintaxis:

<Concatenación sintáctica>::= "(" <operador> { <operador> } ")"

En el siguiente ejemplo se concatena la cadena "Hola", "Mundo" y la producción "suma":

(Hola Mundo %suma)

Negación

Identifica todo lo que no cumpla con el operador que la precede.

Sintaxis:

<Negación>::=<operador> "^"

El siguiente ejemplo identifica cualquier cosa que no sea la cadena "Mundo":

Mundo ^

Borrador

Se encarga de borrar lo que devuelva el operador precedente.

Sintaxis:

<Borrador>::=<operador> /

En este ejemplo borra el resultado de la producción "espacios_en_blanco", cuando esta se cumple:

%espacios_en_blanco /

Categoría léxica

Los tokens contienen una parte con la categoría léxica a la que pertenecen y otra parte con el valor del token. Este operador se encarga de identificar si el token actual tiene la categoría léxica indicada.

Sintaxis:

<Categoría_léxica>::= "&" <identificador>

En este ejemplo se identifican a los tokens con la categoría léxica "número_entero":

&numero_entero

Cadena simple

Es toda cadena que no sea espacios, salto de línea, nombres de producciones o metacaracteres. Se encarga de identificar la cadena indicada.

Ya que en la notación BNF no está definida la negación, se definirá utilizando la notación de expresiones regulares:

Sintaxis:

[^\s] +

En este ejemplo, se declara un operador “Cadena simple” que identificará cadenas que sean iguales a “Bueno”

Bueno

3.2. Diseño del encadenador

La función del encadenador es devolver una secuencia de tokens procesados léxica y sintácticamente, al tener como entrada por parte del usuario los siguientes archivos:

-Un archivo de texto con los patrones léxicos

-Un archivo de texto con los patrones sintácticos (Gramática Libre de Contexto)

-Una cadena a analizar

Esto se muestra gráficamente en el diagrama 3.1:

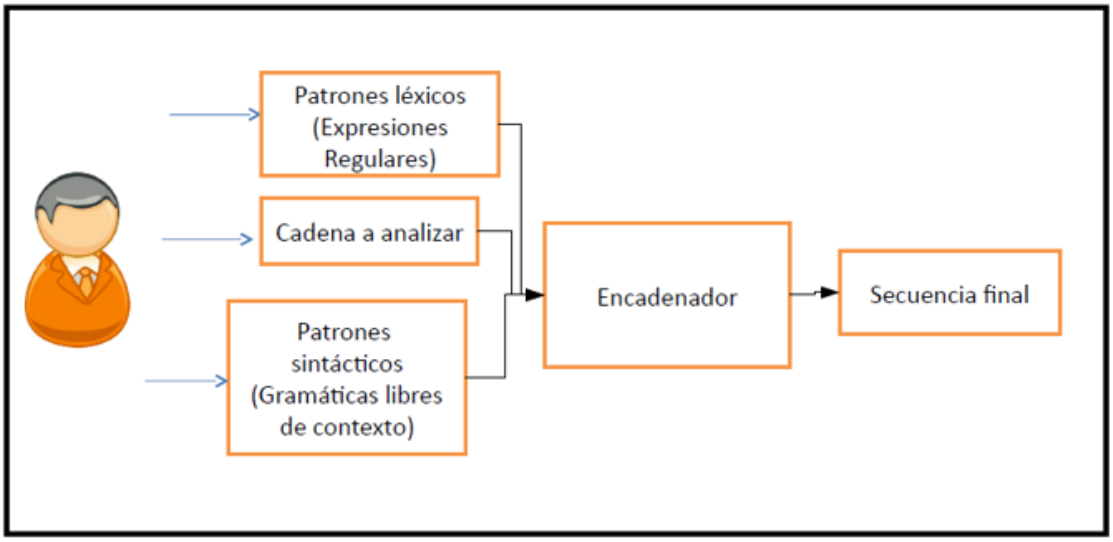


Diagrama 3.1: Interacción de VERT con el usuario

Encadenador

El proceso inicia con una normalización de la cadena a analizar para transformarla en una secuencia de tokens donde cada token contiene un caracter de la secuencia original. Los tokens de esta secuencia están integrados por un nombre de categoría léxica, en este caso es “nulo”, y la segunda parte es la cadena del token, es decir, el valor del token.

Análisis léxico

La secuencia devuelta por el proceso de normalización y la cadena que contiene los patrones léxicos son tomados como entradas para la instancia léxica del analizador sintáctico y devolviendo como salida una secuencia de tokens procesados léxicamente. En esta secuencia es donde se agrupan las partes más pequeñas con sentido en un lenguaje, siendo algunos ejemplos: números, identificadores, símbolos de comparación, operadores aritméticos, etc.

Posteriormente se realiza una normalización léxica, es decir, los caracteres de los tokens reconocidos en la parte léxica son concatenados, ya que se encuentran separados caracter por caracter en un arreglo correspondiente a variables.

Análisis sintáctico

Esta secuencia de tokens normalizados léxicamente y los patrones sintácticos son ingresados como entradas de la instancia sintáctica del analizador sintáctico, dando como resultado una secuencia de tokens procesados sintácticamente.

Este proceso se muestra en el diagrama 3.2.

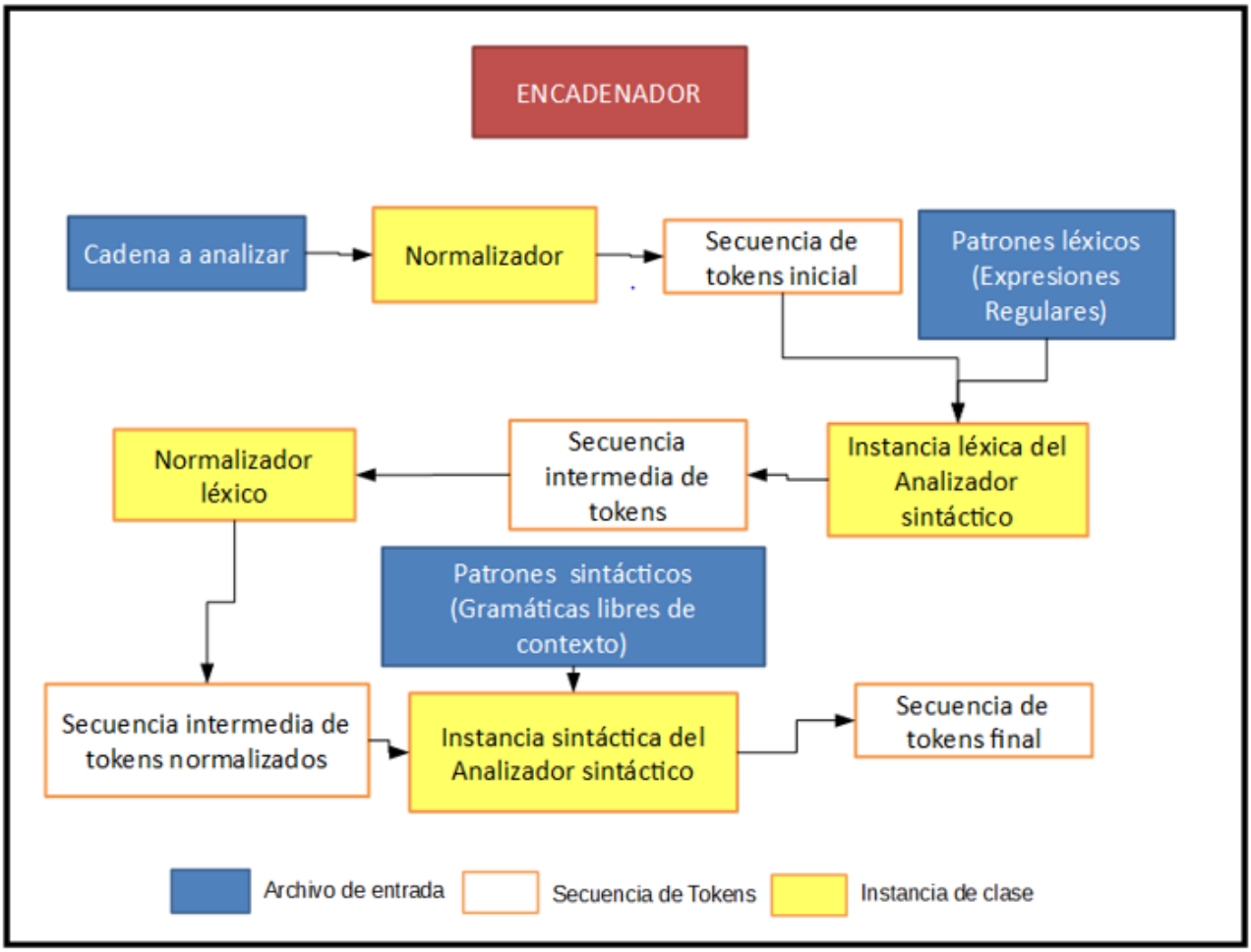


Diagrama 3.2: Estructura interna del Encadenador

3.3. Diseño del analizador sintáctico

Al crear una instancia de un analizador sintáctico, se procesan los patrones (reglas de producción) ingresados como una gramática libre de contexto para generar las instancias correspondientes de entidades sintácticas y unirlas para formar una gramática ejecutable.

Cuando la instancia creada del analizador sintáctico analiza una secuencia de tokens, utiliza la gramática ejecutable construida cuando se creó la instancia, para reconocer las subsecuencia de caracteres que cumplen con esa gramática, y procesarlas para obtener otra secuencia de tokens como salida. Estos dos procesos son ilustrados en el diagrama 3.3.

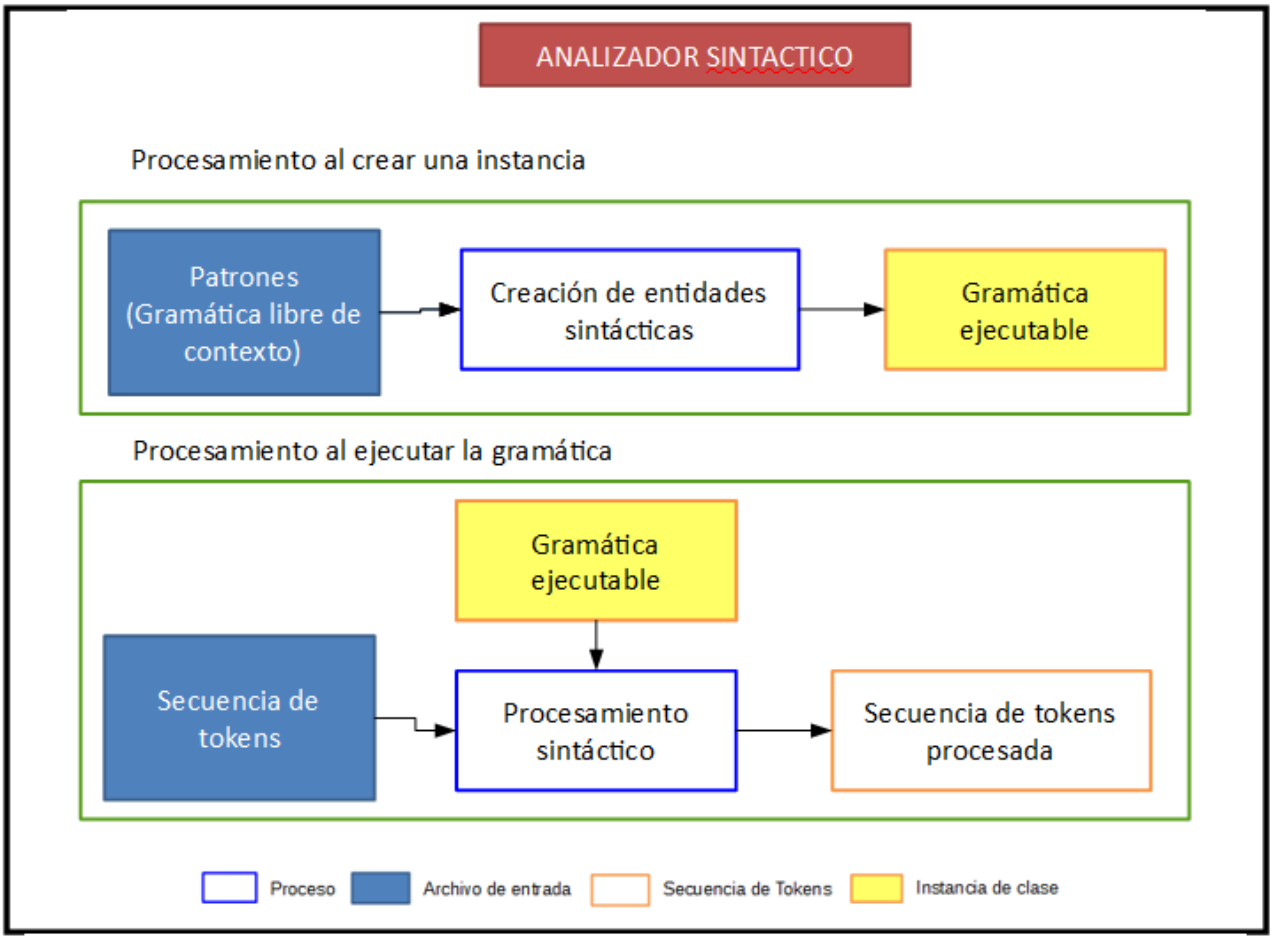


Diagrama 3.3: Estructura interna del analizador sintáctico

Gramática ejecutable

La gramática ejecutable se formará a partir de los operadores sintácticos, de acuerdo a la gramática definida como entrada del Analizador sintáctico. Los operadores sintácticos contienen los comportamientos de las operaciones básicas del lenguaje que implementa VERT. Tales operaciones son: la concatenación, disyunción, negación, cadena simple, categoría léxica, borrador, etc. Los operadores sintácticos son una parte esencial del analizador sintáctico, pues son estos los que se encargan de hacer el análisis tanto léxico como sintáctico, ya que estos análisis utilizan instancias del analizador sintáctico, aprovechando que el conjunto de lenguajes que pueden ser expresados con expresiones regulares son sólo un subconjunto de aquellos que pueden ser expresados con gramáticas libres de contexto. El diseño de los operadores sintácticos se verá más a detalle en la siguiente sección.

3.4. Diseño de operadores

Todos los operadores sintácticos descienden directa o indirectamente del coordinador sintáctico, ya que este operador contiene la secuencia de tokens sobre la cual se está trabajando y el índice de partida, el cual indica en que token se está procesando actualmente de la secuencia, de esta manera las diferentes instancias de los operadores sintácticos pueden saber desde donde deben de empezar a procesar y como manipular el índice de partida.

La herencia está ilustrada en el diagrama 3.4.

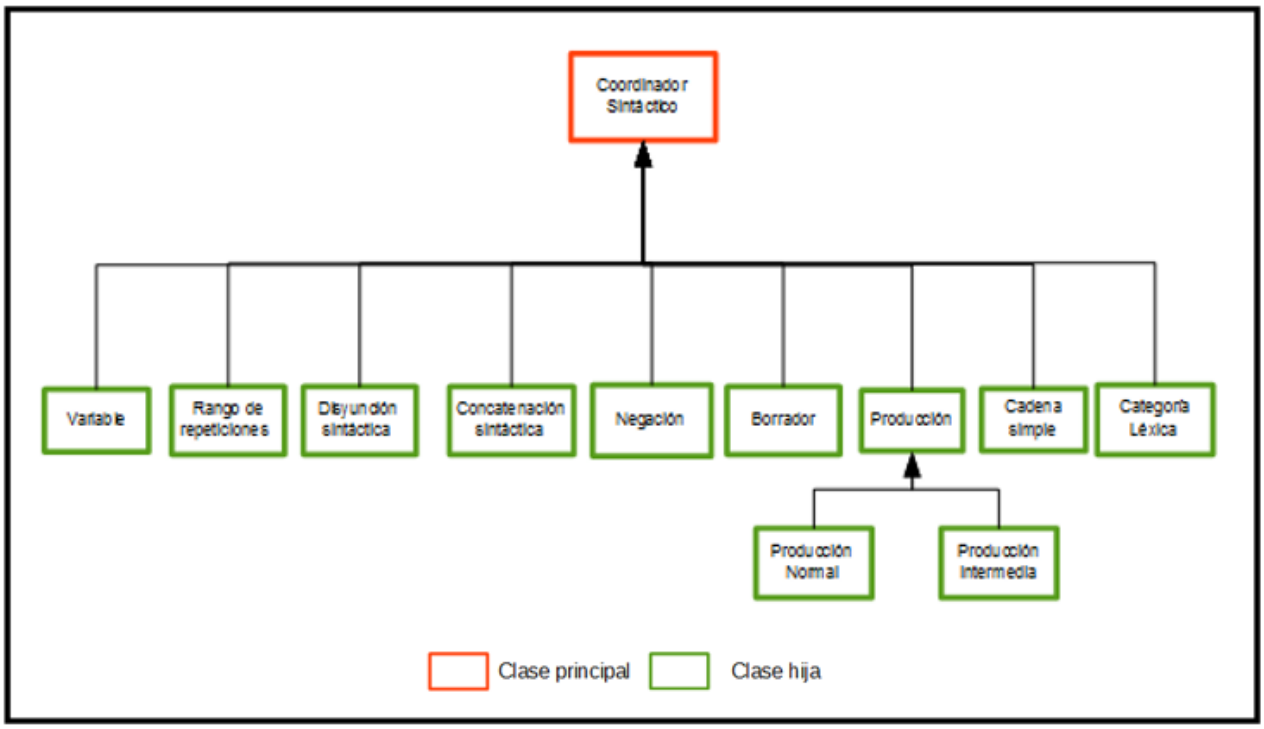


Diagrama 3.4: Estructura de herencia de los componentes

Operador sintáctico: Variable

Este operador utiliza como entrada otro objeto sintáctico. En caso de ejecutar el objeto de entrada y que este devuelva “nulo”, la instancia del operador Variable devolverá de igual manera “nulo”. En caso de que el objeto de entrada devuelva algo diferente a “nulo”, la instancia del operador Variable insertara este resultado en el último elemento de la pila que contiene las variables de las producciones en ejecución.

El funcionamiento del presente operador está ilustrado en el diagrama 3.5.

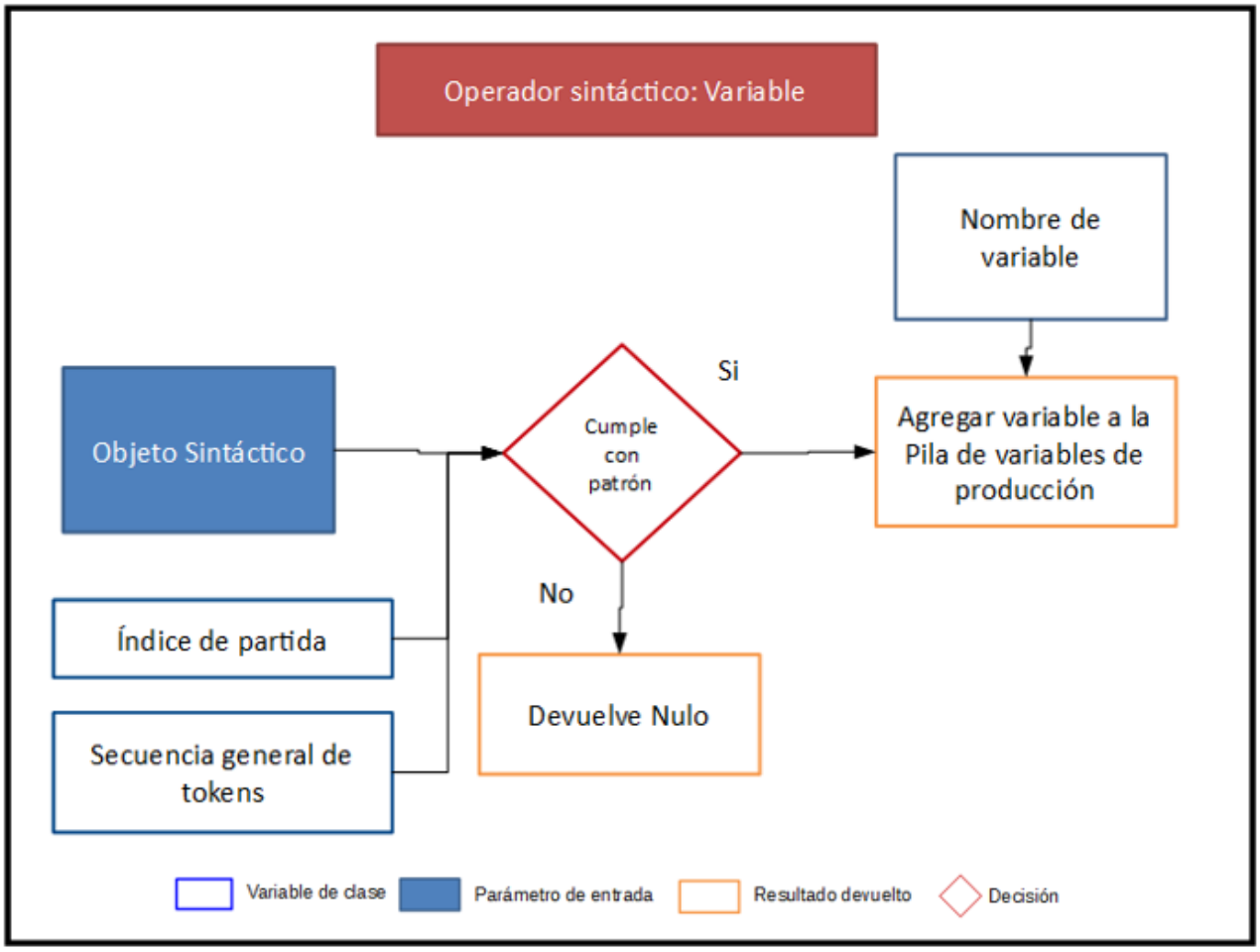


Diagrama 3.5: Funcionamiento del operador sintáctico Variable

Operador sintáctico: Rango de repeticiones

Este operador toma como entrada otro objeto sintáctico, un número mínimo y un número máximo de repeticiones. Este operador devuelve "nulo" cuando el número de veces que el objeto sintáctico de entrada se ejecuta exitosamente un número de veces que no se encuentre entre en número mínimo y máximo de repeticiones que se dieron como entrada, en caso contrario, devuelve el conjunto de todos los resultados exitosos que tuvo el objeto de entrada.

El funcionamiento del presente operador está ilustrado en el diagrama 3.6.

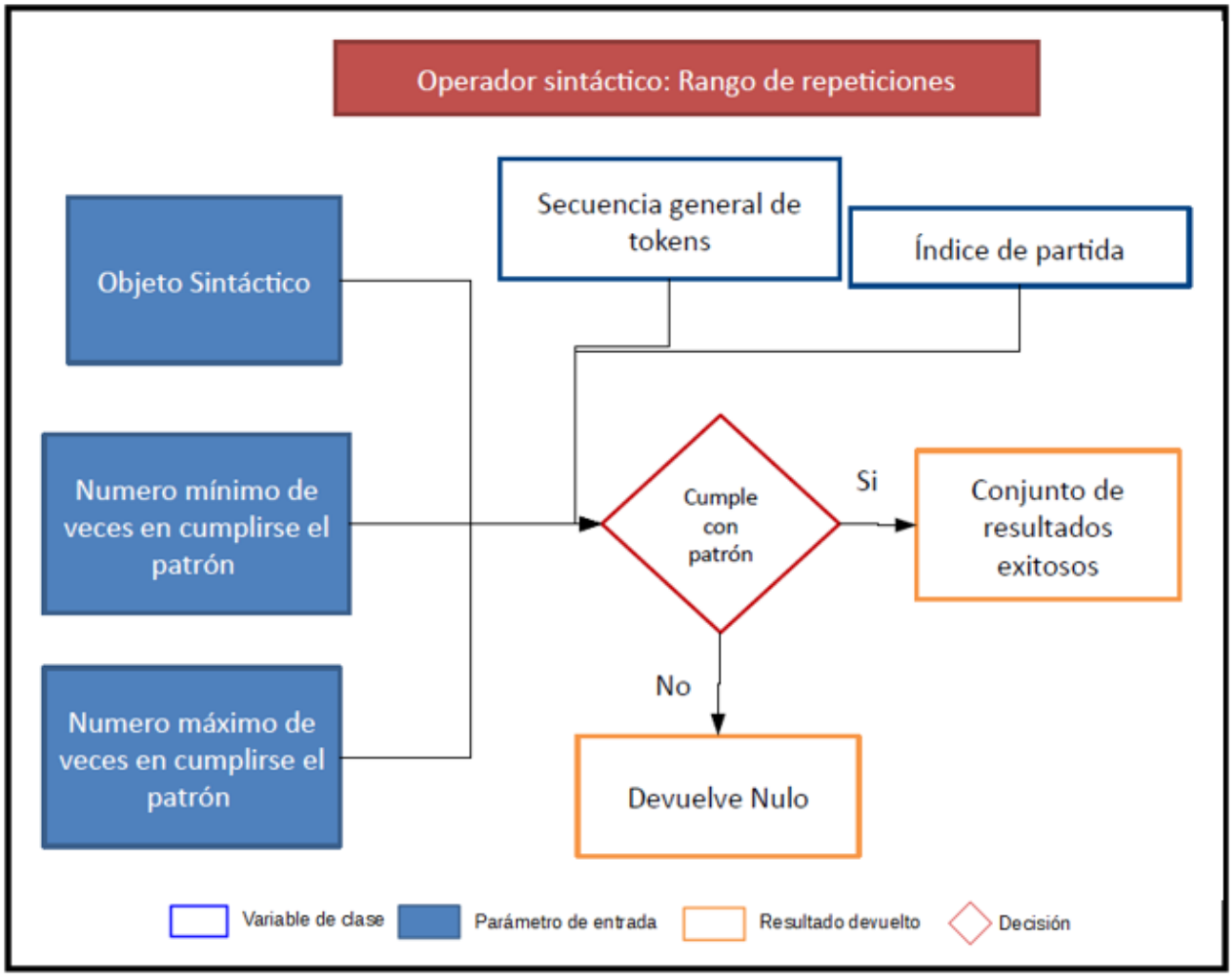


Diagrama 3.6: Funcionamiento del operador sintáctico Rango de Repeticiones

Operador sintáctico: Disyunción sintáctica

Este operador toma n objetos sintácticos de entrada y devuelve “nulo” si ningún objeto de entrada es capaz de reconocer una subsecuencia de tokens que se encuentren delante del token indicado por el índice de partida, y en caso de que algún objeto de la disyunción reconozca exitosamente una subsecuencia, la disyunción devolverá el resultado del objeto exitoso.

El funcionamiento del presente operador está ilustrado en el diagrama 3.6.

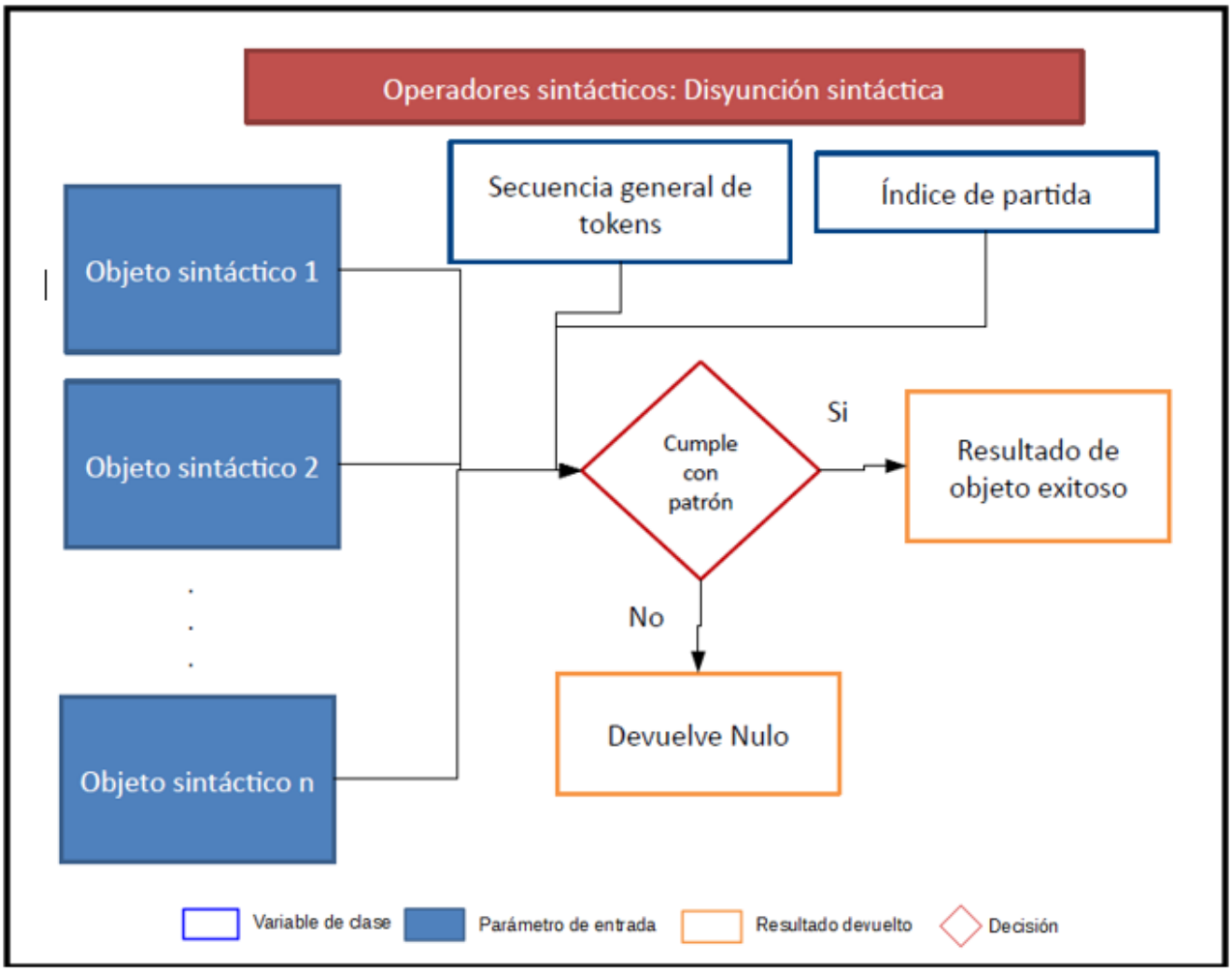


Diagrama 3.7: Funcionamiento del operador Disyunción sintáctica

Operador sintáctico: Concatenación sintáctica

Tiene como entradas n objetos sintácticos que, al efectuarse la concatenación de ellos, se ejecutan secuencialmente. Este operador devuelve “nulo” cuando alguno de los objetos que tiene de entrada tiene como salida “nulo”, y es exitoso cuando todos los objetos sintácticos de entrada tienen éxito, en este caso devuelve el conjunto de resultados exitosos respetando el orden de los objetos.

El funcionamiento del presente operador está ilustrado en el diagrama 3.8.

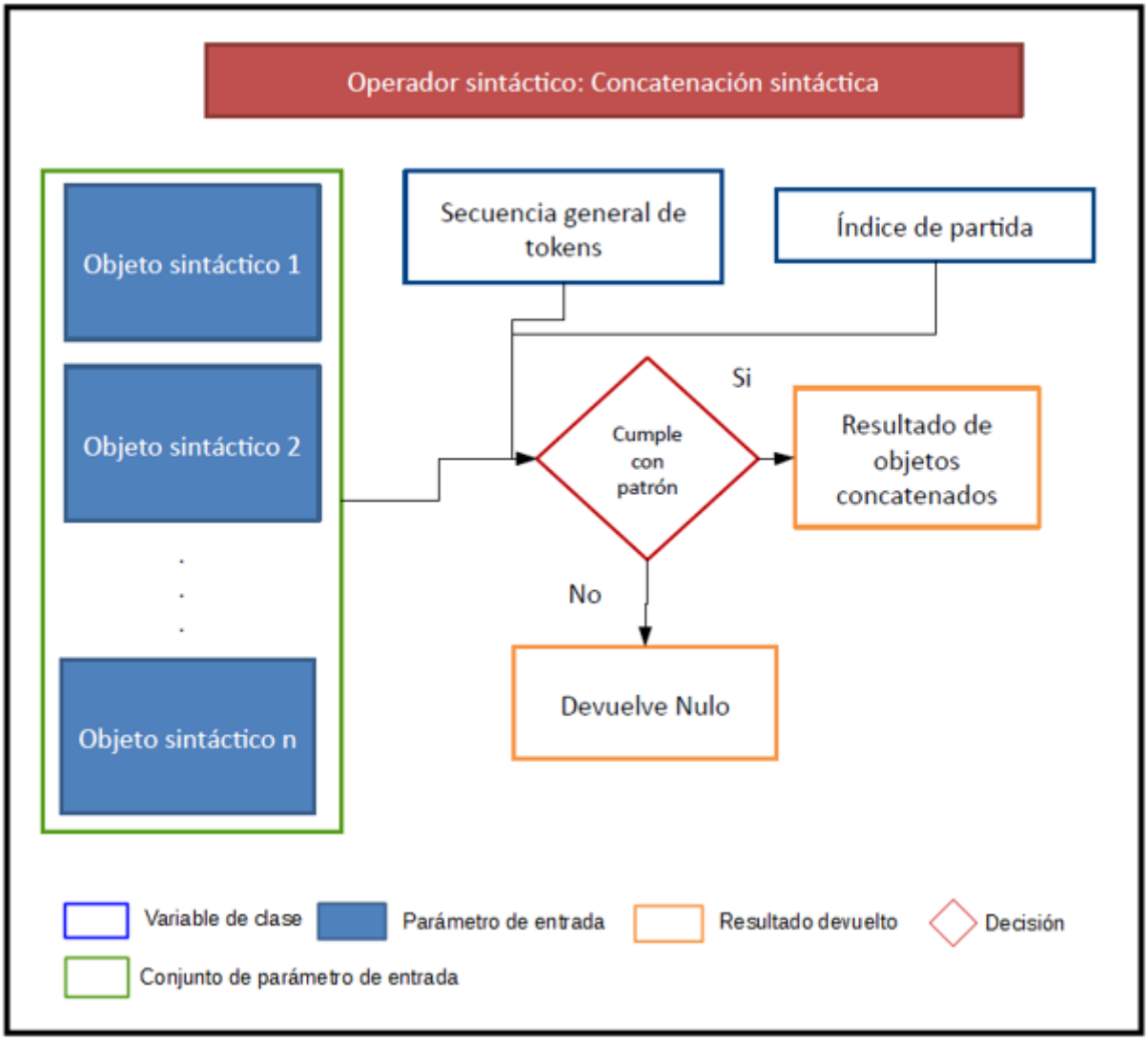


Diagrama 3.8: Funcionamiento del operador Concatenación sintáctica

Operador sintáctico: Negación

Este operador toma como entrada otro objeto sintáctico y devuelve “nulo” si el objeto de entrada tuvo éxito en su ejecución, en caso contrario, devuelve la cadena del token al que apunta el índice de partida.

El funcionamiento del presente operador está ilustrado en el diagrama 3.9.

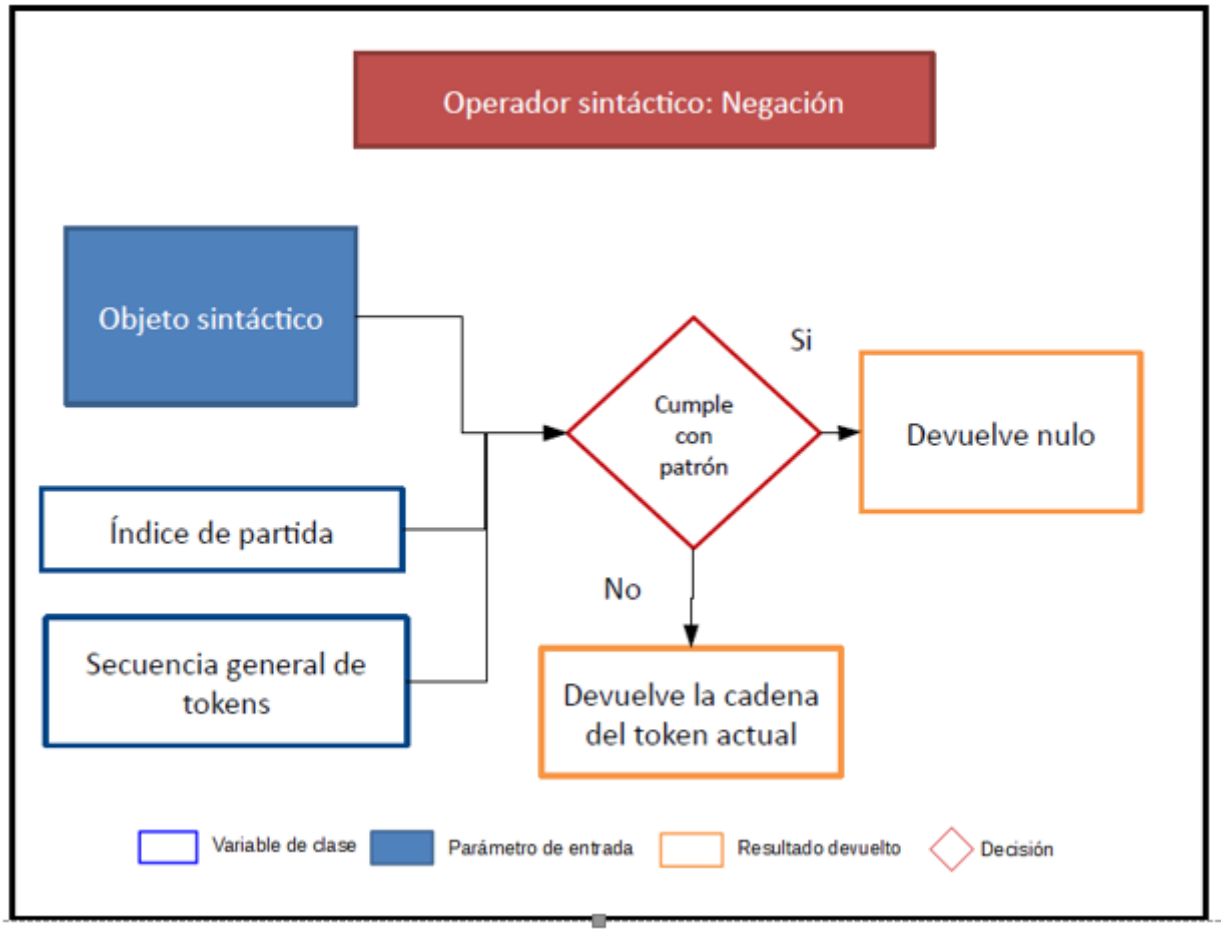


Diagrama 3.9: Funcionamiento del operador sintáctico Negación

Operador Sintáctico: Borrador

Este operador utiliza de entrada un objeto sintáctico, siendo su salida “nulo” cuando el objeto de entrada genera “nulo” como salida, pero el operador devuelve un objeto especial que será ignorado por el objeto que contenga una instancia del operador Borrador.

El funcionamiento del presente operador está ilustrado en el diagrama 3.10.

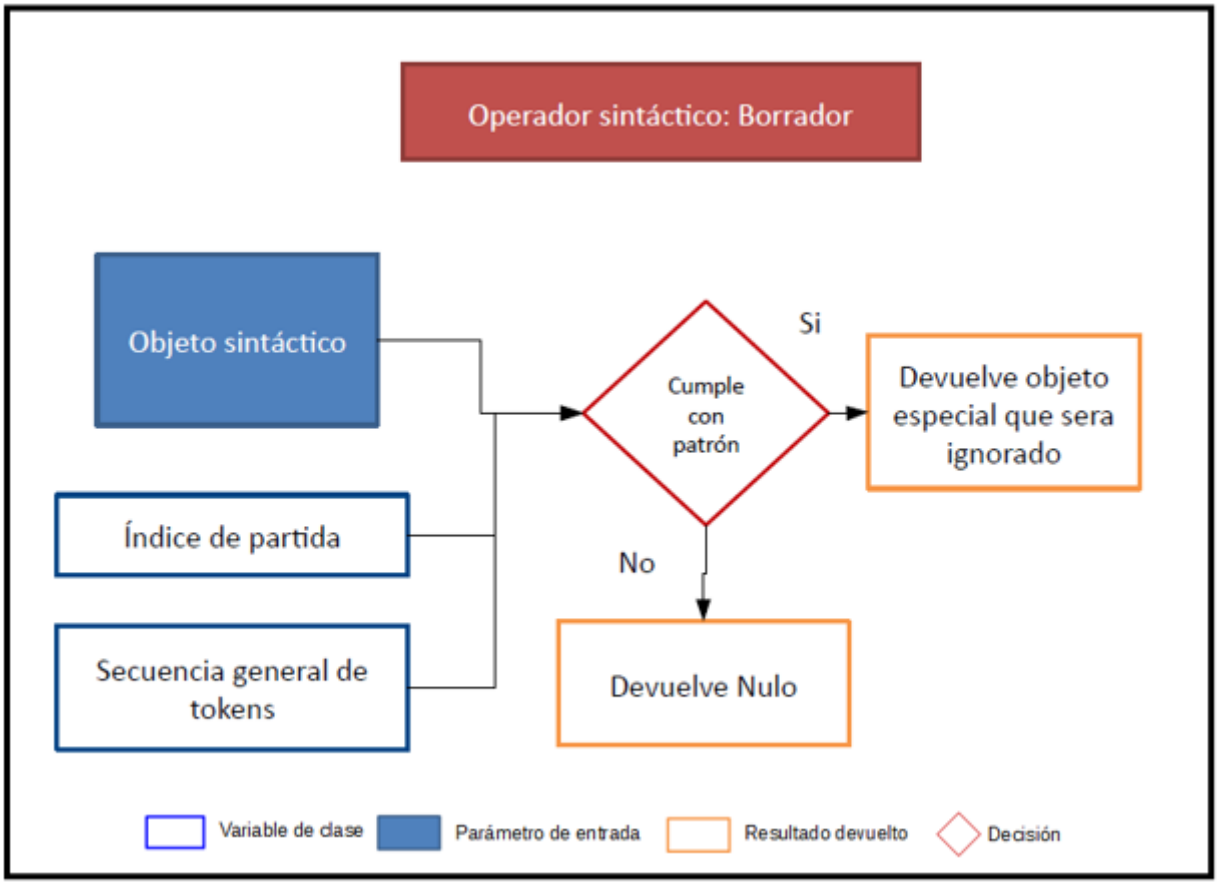


Diagrama 3.10: Funcionamiento del operador sintáctico Borrador

Descendencia de Producción

Los operadores “Producción normal” y “Producción intermedia” heredan el comportamiento del operador Producción que a su vez hereda del operador Concatenación sintáctica. Los operadores Producción normal y Producción intermedia difieren en su ejecución, pero comparte el preprocesamiento antes de ejecutarse.

La estructura de la herencia anteriormente mencionada está ilustrado en el diagrama 3.11.

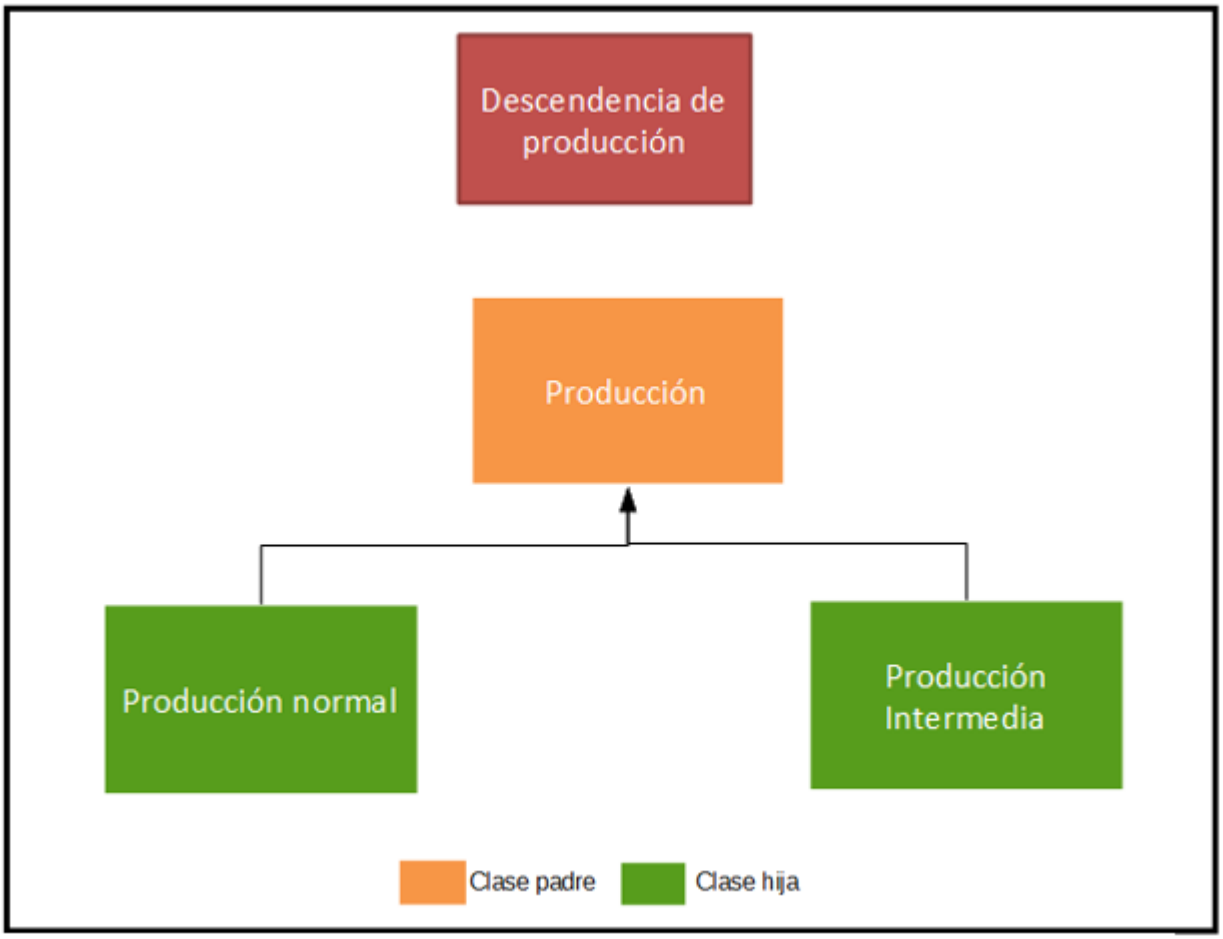


Diagrama 3.11: Relación de herencia de producciones

Operador sintáctico: Producción normal

Tiene un comportamiento similar a la Concatenación sintáctica, toma n objetos sintácticos como entrada. Ejecuta uno por uno los objetos en el orden que se definió la producción, dando como resultado "nulo" si alguno de los objetos devolvió "nulo" después de ejecutarse, y en caso de que todos los objetos se hayan ejecutado con éxito, la instancia de Producción normal dará como salida un token compuesto de un lado por el nombre de categoría léxica, en este caso el nombre de la Producción normal, y por el otro el conjunto de resultados devueltos por los objetos que contiene, respetando el orden.

El funcionamiento del presente operador está ilustrado en el diagrama 3.12.

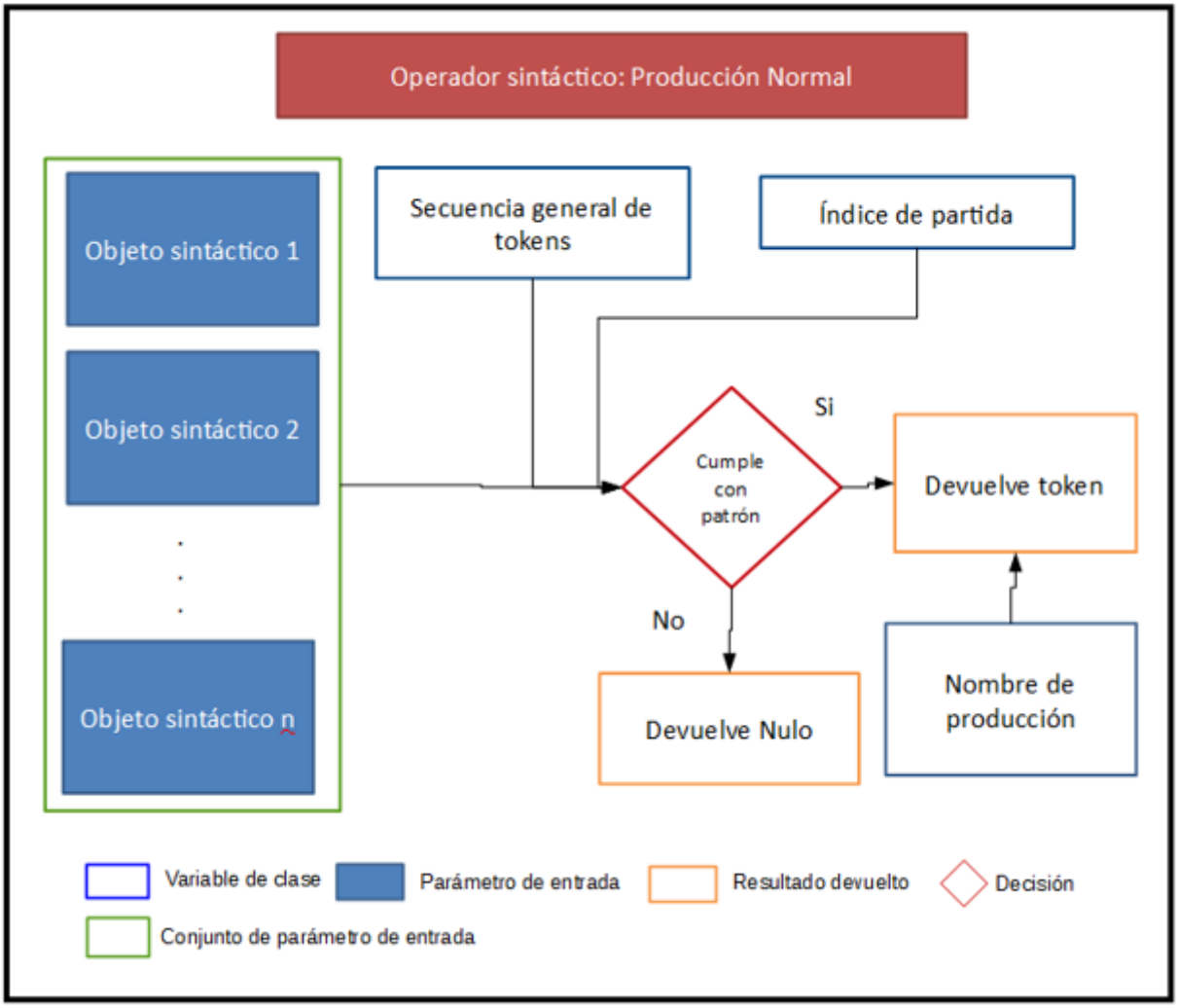


Diagrama 3.12: Funcionamiento del operador sintáctico Producción normal

Operador sintáctico: Producción intermedia

Este operador se comporta prácticamente igual que la Concatenación sintáctica, con la diferencia de que este operador tiene nombre y puede ser invocado y referenciado. Esta producción se utiliza para la producción de inicio que es la que comienza la ejecución.

El funcionamiento del presente operador está ilustrado en el diagrama 3.13.

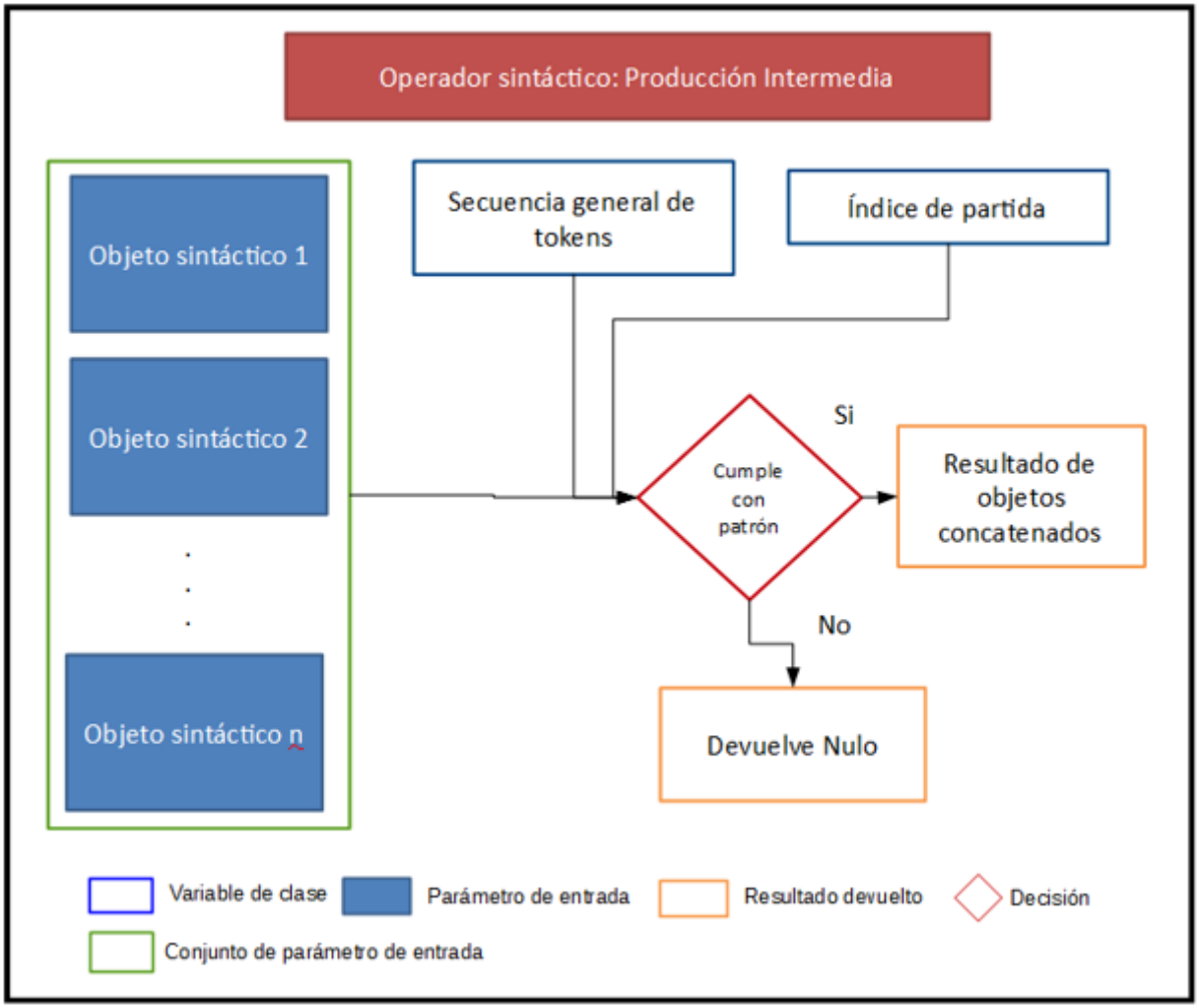


Diagrama 3.13: Funcionamiento del operador sintáctico Producción intermedia

Operador sintáctico: Categoría léxica

Como entrada tiene el nombre de categoría léxica contra la cual comparara dicha sección del token que apunte el índice de partida. Su resultado será “nulo” si el nombre de la categoría léxica del token actual es diferente al nombre que contiene el operador, en caso contrario, devolverá la cadena del token actual.

El funcionamiento del presente operador está ilustrado en el diagrama 3.14.

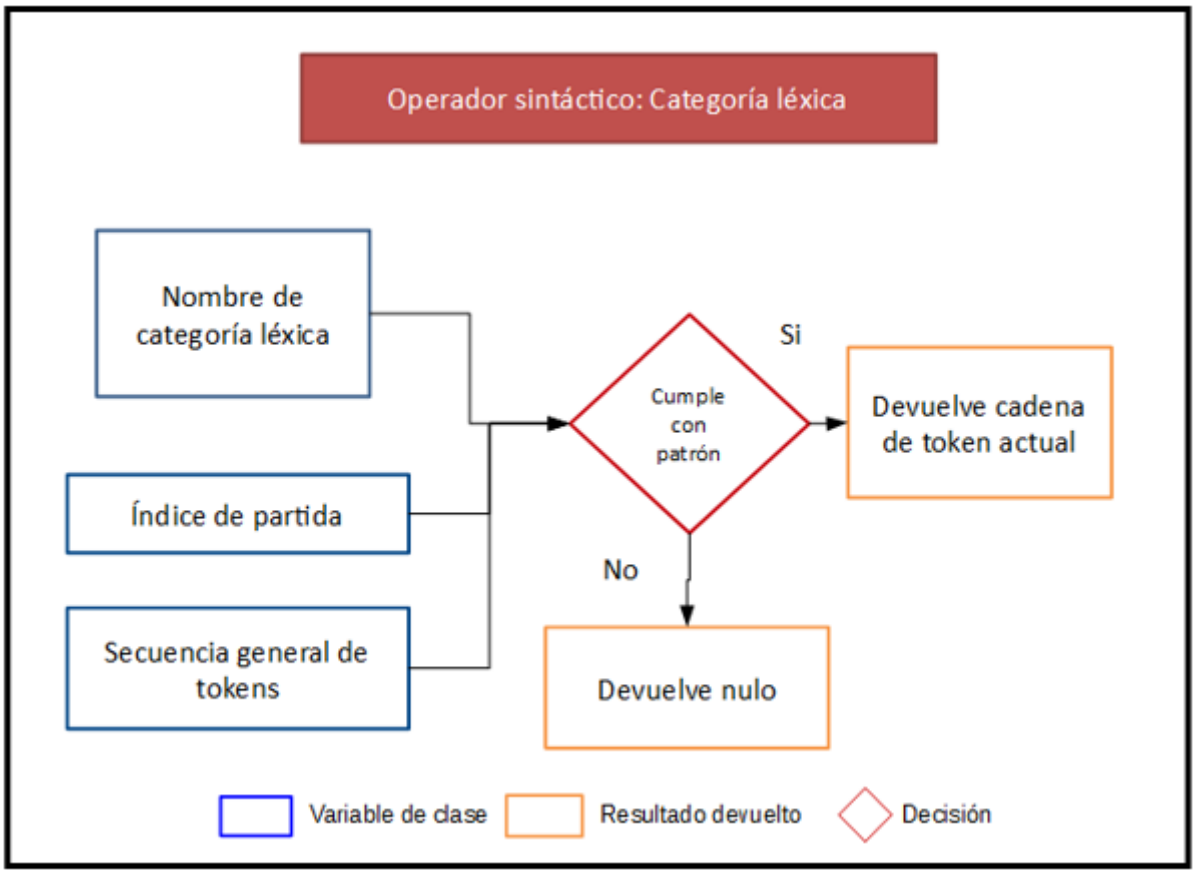


Diagrama 3.14: Funcionamiento del operador Categoría léxica

Operador sintáctico: Cadena simple

Este operador acepta como entrada una cadena de caracteres y la compara con la sección de contenido del token actual al que apunta el índice de partida. El operador dará un "nulo" si no coinciden las cadenas, tanto la que contiene el operador como la del token actual, pero si las cadenas coinciden, devuelve la cadena que contiene el operador.

El funcionamiento del presente operador está ilustrado en el diagrama 3.15.

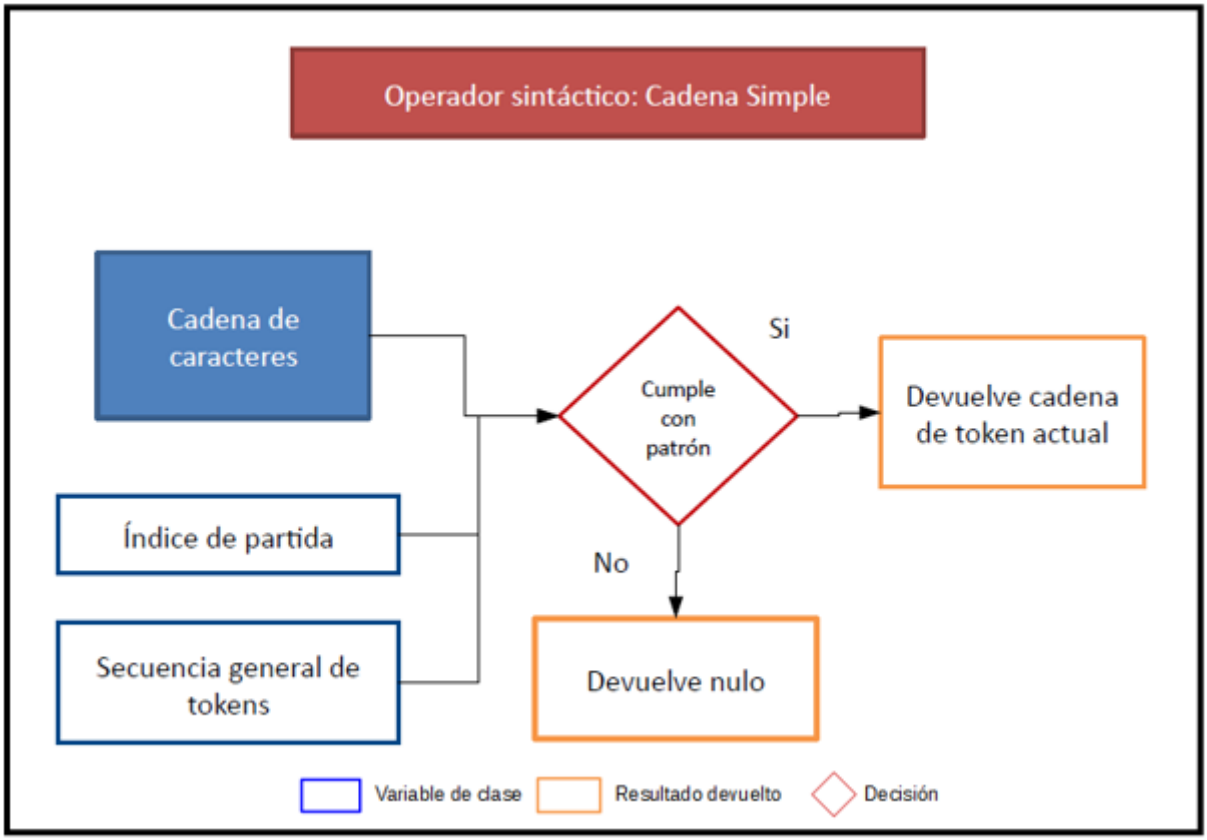


Diagrama 3.15: Funcionamiento del operador Cadena simple

3.5. Diseño del algoritmo de optimización de disyunción sintáctica

Ejecución secuencial simple

Realiza el recorrido de las diferentes opciones de la disyunción hasta que uno de elementos acepta la subsecuencia de tokens identificados por el patrón del elemento o ningún elemento de la disyunción es capaz de reconocer la subsecuencia siguiente al índice de partida.

Elemento 1	Elemento 2	Elemento 3	Elemento 4	Elemento 5	Elemento ….	Elemento N

Tabla 3.1: Inicio de ejecución secuencial simple

Recorrido intento 1

En el intento 1, el elemento 1 no es capaz de reconocer una subsecuencia de tokens iniciando desde el índice de partida.

Elemento 1	Elemento 2	Elemento 3	Elemento 4	Elemento 5	Elemento ….	Elemento N
x						

Tabla 3.2: Prueba con Elemento 1 en la ejecución secuencial simple

Recorrido intento 2

Al no tener éxito en el intento uno continua con el siguiente elemento. En el intento 2, el elemento 2 no es capaz de reconocer una subsecuencia de tokens iniciando desde el índice de partida.

Elemento 1	Elemento 2	Elemento 3	Elemento 4	Elemento 5	Elemento ….	Elemento N
x	x					

Tabla 3.3: Prueba con Elemento 2 en la ejecución secuencial simple

Recorrido intento 3

Elemento 1	Elemento 2	Elemento 3	Elemento 4	Elemento 5	Elemento ….	Elemento N
x	x	x				

Tabla 3.3: Prueba con Elemento 3 en la ejecución secuencial simple

Recorrido intento 4

Elemento 1	Elemento 2	Elemento 3	Elemento 4	Elemento 5	Elemento ….	Elemento N
x	x	x	x			

Tabla 3.4: Prueba con Elemento 4 en la ejecución secuencial simple

Recorrido intento 5

Elemento 1	Elemento 2	Elemento 3	Elemento 4	Elemento 5	Elemento ….	Elemento N
x	x	x	x	x		

Tabla 3.5: Prueba con Elemento 5 en la ejecución secuencial simple

Recorrido intento n-1

Elemento 1	Elemento 2	Elemento 3	Elemento 4	Elemento 5	Elemento ….	Elemento N
x	x	x	x	x	x	

Tabla 3.6: Prueba con Elemento n-1 en la ejecución secuencial simple

Recorrido intento N

Una posible resultado es que, al llegar al elemento n este reconozca una subsecuencia desde el índice de partida, la disyunción devolverá el resultado del elemento n cuando el recorrido llega hasta el elemento n siendo n cualquier elemento, este sí es capaz de reconocer una subsecuencia desde el índice de partida devolviendo, la disyunción.

Elemento 1	Elemento 2	Elemento 3	Elemento 4	Elemento 5	Elemento ….	Elemento N
x	x	x	x	x	x	Correcto

Tabla 3.7: Prueba con Elemento N en la ejecución secuencial simple

Recorrido completo no exitoso

El otro posible escenario se presenta cuando ninguno de los elementos contenidos en la disyunción puede reconocer una subsecuencia, en este caso, el resultado que devolverá será un "nulo" el cual simboliza que el reconocimiento de las disyunción no tuvo éxito.

Elemento 1	Elemento 2	Elemento 3	Elemento 4	Elemento 5	Elemento ….	Elemento N
x	x	x	x	X	x	x

Tabla 3.8: Prueba con Elemento N en la ejecución secuencial simple en segundo escenario

Ejecución optimizada

Funcionamiento del algoritmo:

Hace un recorrido seleccionando los elementos que posiblemente puedan reconocer una subsecuencia, esta selección se hace con base a los símbolos con los que inicia el reconocimiento de cada uno de los elementos de la disyunción los cuales se guardan en una lista conformada por pares Símbolo -> sub conjunto de elementos de la disyunción que inician su reconocimiento con dicho símbolo.

Elemento 1	Elemento 2	Elemento 3	Elemento 4	Elemento 5	Elemento ….	Elemento N

Tabla 3.9: Conjunto de elementos de la disyunción

Explicación

Teniendo el símbolo del índice de partida busca en la lista de símbolos iniciales seleccionando el subconjunto de elementos de la disyunción que empiezan con ese símbolo

Elemento 2	Elemento N

Tabla 3.10: Elementos candidatos en ejecución optimizada

Recorrido intento 1

En el intento 1, el elemento 2 no es capaz de reconocer una subsecuencia de tokens iniciando desde el índice de partida.

Elemento 2	Elemento N
x	

Tabla 3.11: Prueba de elemento 2 en ejecución optimizada

Recorrido intento 2

Al no tener éxito en el intento uno continua con el siguiente elemento .En el intento 2, el elemento 2 no es capaz de reconocer una subsecuencia de tokens iniciando desde el índice de partida.

Elemento 2	Elemento N
x	Correcto

Tabla 3.12: Prueba de Elemento N en ejecución optimizada

Recorrido del subconjunto no exitoso

El otro posible escenario se presenta cuando ninguno de los elementos contenidos en la disyunción puede reconocer una subsecuencia, en este caso, el resultado que devolverá será un "nulo" el cual simboliza que el reconocimiento de las disyunción no tuvo éxito.

Comparación entre algoritmos

Ventajas	Algoritmo simple	Algoritmo optimizado
No necesita preprocesamiento antes de ejecutarse	X	
No consume memoria adicional para la creación de una lista	X	
Baja complejidad	X	
Menor tiempo de ejecución		X

Tabla 3.13: Comparación de algoritmos

El algoritmo simple tiene una serie de ventajas respecto al algoritmo optimizado, como se puede ver en la Tabla 3.13, como los son: su baja complejidad, ahorro de memoria y la falta de preprocesamiento antes de su ejecución, sin embargo el algoritmo optimizado cuenta con una ventaja con mayor peso que las anteriores, y esta es un menor tiempo de ejecución. La velocidad de ejecución de los análisis léxico y sintáctico son una prioridad cuando se trata de archivos de gran tamaño o gramáticas complejas.

Capítulo 4: Implementación de la nueva herramienta

Introducción

En este capítulo se explica paso a paso la ejecución del analizador léxico y sintáctico, plasmando el funcionamiento del programa, a partir de una serie de pasos dependientes que conforman la estructura Implementada en lenguaje Ruby. Cabe mencionar que el lenguaje de programación Ruby fue elegido para la implementación por la facilidad de procesamiento y manipulación de cadenas de texto, y por ser un lenguaje interpretado, lo cual facilita la depuración del código y esta es una característica que se desea en la nueva herramienta a implementar.

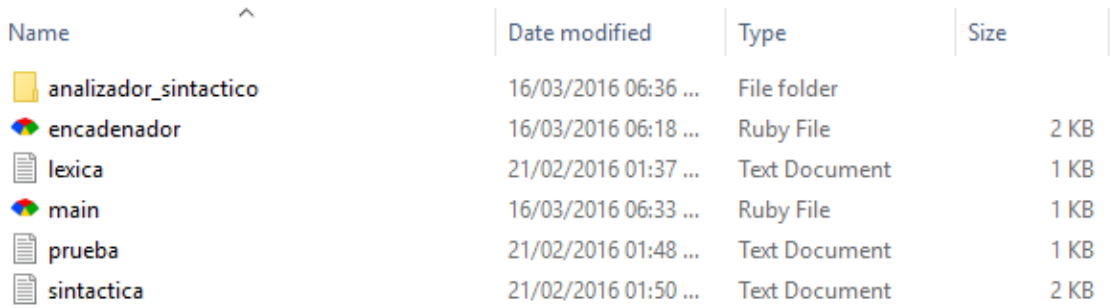
4.1. Organización de archivos

La implementación se llevó a cabo con un programa principal llamado “main.rb” en el cual el usuario introduce las rutas de los archivos que contienen los patrones léxicos y sintácticos, y el archivo a analizar.

En la misma carpeta se encuentra “encadenador.rb” que es el encargado de unir el proceso léxico y sintáctico.

En una subcarpeta llamada “analizador_sintactico” se encuentran los archivos de los operadores sintácticos y el analizador sintáctico.

Esto se ve más claramente en la Imagen 4.1.



Name	Date modified	Type	Size
analizador_sintactico	16/03/2016 06:36 ...	File folder	
encadenador	16/03/2016 06:18 ...	Ruby File	2 KB
lexica	21/02/2016 01:37 ...	Text Document	1 KB
main	16/03/2016 06:33 ...	Ruby File	1 KB
prueba	21/02/2016 01:48 ...	Text Document	1 KB
sintactica	21/02/2016 01:50 ...	Text Document	2 KB

Imagen 4.1: Organización de los archivos

En la subcarpeta “analizador_sintactico” se encuentran los archivos (Imagen 4.2):

-“analizador_sintactico.rb” que contiene el comportamiento del analizador sintáctico.

-“operadores_sintacticos.rb” que contiene las clases de los distintos operadores sintácticos que utiliza el analizador sintáctico.

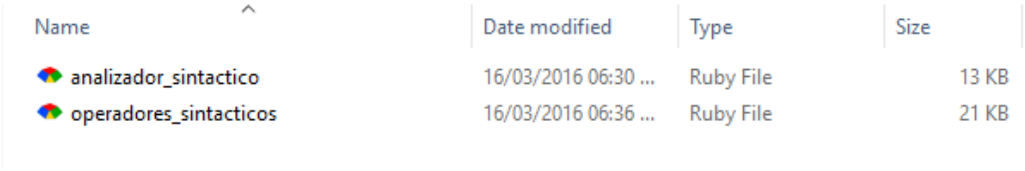


Imagen 4.2: Organización de archivos del analizador sintáctico

4.2. Implementación del Encadenador y los operadores

Implementación del Encadenador “main.rb” (Imagen 4.3)

```
require_relative  'encadenador'

encadenamiento = Encadenador.new(
  [

    'lexica.txt',

    'sintactica.txt'

  ]
)

cadena_a_analizar = File.open('prueba.txt', 'r') {  |archivo|

                                                    archivo.read

                                                }

encadenamiento.recepcion_de_cadena_a_analizar( cadena_a_analizar )
arbol = encadenamiento.encadenamiento_de_gramaticas
```

Imagen 4.3: Código de implementación de VERT

El programa invoca a la instancia de la clase encadenador y se le da como argumento:

-“lexica.txt” que contiene los patrones léxicos.

-“sintáctica.txt” que contiene los patrones sintácticos

-“prueba.txt” que contiene el archivo a analizar

Posteriormente se ejecuta el análisis con el método “encadenamiento_de_gramaticas”

Implementación del Encadenador (Imagen 4.4)

```
class Encadenador

  require_relative  'analizador_sintactico/analizador_sintactico'

  def initialize( parametros_de_gramaticas ){...}

  def recepcion_de_cadena_a_analizar  cadena_a_analizar{...}

  def encadenamiento_de_gramaticas{...}

end
```

Imagen 4.4: Código de implementación de la clase Encadenador

El encadenador llama al archivo “analizador_sintactico.rb” para poder generar instancias de la clase Analizador_sintactico.

El método “initialize” se encarga de hacer los procesos necesarios cuando se crea una instancia de la clase Encadenador.

El método “recepción_de_cadena_a_analizar” recibe la cadena que se va analizar.

El método “encadenamiento_de_gramaticas” ejecuta el procesamiento léxico seguida del sintáctica.

Implementación del analizador sintáctico (Imagen 4.5)

```
Enlace = Struct.new( :no_terminal )
class Analizador_sintactico

  require_relative 'operadores_sintacticos'
  Caracter_escapado = Struct.new( :caracter )
  Almacenador_temporal = Struct.new( :simbolo, :indice )

  def initialize( gramatica, grupos ){...}

  def creacion_de_producciones_de_gramatica_de_reconocimiento{...}

  def pasada_simple_de arreglo_a_analizar{...}

end
```

Imagen 4.5: Código de implementación del Analizador sintáctico

El analizador sintáctico llama el archivo “operadores_sintacticos” que contiene los operadores sintácticos que utilizara para transformar los patrones dados, en una gramática ejecutable por Ruby.

Implementación de los operadores sintácticos (Imagen 4.6)

```
class Coordinador_sintactico

  def initialize( producciones ){...}
  def nueva_secuencia_de_tokens secuencia_de_tokens{...}
  def tamano_de_secuencia_de_tokens{...}
  def token_actual{...}
  def avanza_token{...}
  def actualizar_indice_con valor_a_actualizar{...}
  def indice_actual{...}

end
```

Imagen 4.6: Código de implementación del Coordinador sintáctico

El coordinador sintáctico es el encargado de llevar un control sobre el token actual que se está procesando en la secuencia de tokens.

```
class Variable < Coordinador_sintactico

  def initialize( nombre_de_variable, elemento_que_dara_el_valor_a_la_variable ){...}
  def verificar_archivo{...}
  def busqueda_de_simbolos_iniciales{...}
  def busqueda_inicial_de_simbolos_iniciales{...}
  def creacion_de_enlaces{...}

end
```

Imagen 4.7: Código de implementación del operador sintáctico Variable

La clase Variable (Imagen 4.7) se encarga de guardar el resultado de los operadores sintácticos que englobe.

```
class Rango_de_repeticiones < Coordinador_sintactico

  attr_reader :numero_1

  def initialize contenido, numero_1, numero_2, tiene_coma{...}
  def busqueda_inicial_de_simbolos_iniciales{...}
  def busqueda_de_simbolos_iniciales{...}
  def creacion_de_enlaces{...}
  def verificar_archivo{...}

  alias_method :verificar_archivo_en_disyuncion, :verificar_archivo

end
```

Imagen 4.8: Código de implementación del operador sintáctico Rango de repeticiones

La clase rango de repeticiones (Imagen 4.8) acepta una subsecuencia si se cumple que los operadores que englobe tengan éxito un número de veces que se encuentre en el rango indicado.

```
class Disyuncion_sintactica < Coordinador_sintactico

  def initialize( contenido ){...}
  def busqueda_inicial_de_simbolos_iniciales{...}
  def busqueda_de_simbolos_iniciales{...}
  def creacion_de_enlaces{...}
  def verificar_archivo{...}

end
```

Imagen 4.9: Código de implementación del operador Disyunción sintactica

Los métodos “búsqueda_inicial_de_simbolos_iniciales” y “búsqueda_de_simbolos_iniciales” (Imagen 4.9), que están en casi todas las clases de operadores sintácticos, son necesarios para el algoritmo de optimización que implementa la disyunción sintáctica.

```
class Concatenacion_sintactica < Coordinador_sintactico

  def initialize( contenido ){...}
  def busqueda_inicial_de_simbolos_iniciales{...}
  def busqueda_de_simbolos_iniciales{...}
  def creacion_de_enlaces{...}
  def verificar_archivo{...}

  alias_method :verificar_archivo_en_disyuncion, :verificar_archivo

end
```

Imagen 4.10: Código de implementación del operador sintáctico Concatenación sintactica

La concatenación sintáctica (Imagen 4.10) es la encargada de verificar el cumplimiento de n operadores lógicos, que contenga, de manera secuencial.

```
class Negacion < Coordinador_sintactico

  def initialize( contenido ){...}
  def busqueda_inicial_de_simbolos_iniciales{...}
  def busqueda_de_simbolos_iniciales{...}
  def creacion_de_enlaces{...}
  def verificar_archivo{...}

  alias_method :verificar_archivo_en_disyuncion, :verificar_archivo

end
```

Imagen 4.11: Código de implementación del operador sintáctico Negación

La negación (Imagen 4.11) tiene un comportamiento inverso al de la mayoría de operadores. Tiene éxito cuando el operador sintáctico que contiene fracasa al intentar reconocer una subsecuencia de tokens.

```
class Borrador < Coordinador_sintactico

  def initialize( contenido ){...}
  def busqueda_inicial_de_simbolos_iniciales{...}
  def busqueda_de_simbolos_iniciales{...}
  def creacion_de_enlaces{...}
  def verificar_archivo{...}

  alias_method :verificar_archivo_en_disyuncion, :verificar_archivo

end
```

Imagen 4.12: Código de implementación del operador sintáctico Borrador

Esta clase Borrador (Imagen 4.12) como resultado un símbolo especial que indica que no debe de ser tomado en cuenta por el operador que lo contiene, esto en caso de que el operador que contiene la instancia del Borrador tenga éxito, en caso contrario, regresa un nulo que indica que no tuvo éxito.

```
class Produccion < Concatenacion_sintactica

  attr_reader :nombre_de_produccion

  def initialize nombre_de_produccion, contenido_de_produccion{...}
  def recorrido_completo_de_busqueda_de_simbolos_iniciales{...}
  def busqueda_de_simbolos_iniciales{...}

end
```

Imagen 4.13: Código de implementación del operador sintáctico Producción

El comportamiento del operador Producción (Imagen 4.13) es muy similar al de Concatenación_sintactica, pero la diferencia está en que la Producción cuenta con un nombre.

```
class Produccion_normal < Produccion

  def verificar_archivo{...}

  alias_method :verificar_archivo_en_disyuncion, :verificar_archivo

end
```

Imagen 4.14: Código de implementación del operador sintáctico Producción normal

Una Produccion_normal (Imagen 4.14) devuelve un token, esta es la principal diferencia con el resto de operadores sintácticos. El token está integrado por 2 partes: el nombre_de_categoria que es el nombre de la producción, y el resultado obtenido de ejecutar los operadores que contiene.

Este operador devuelve un token en caso de tener éxito, sino devuelve un nulo.

```
class Produccion_intermedia < Produccion

  def verificar_archivo{...}

  alias_method :verificar_archivo_en_disyuncion, :verificar_archivo

end
```

Imagen 4.15: Código de implementación del operador sintáctico Producción intermedia

La diferencia entre Produccion_normal (Imagen 4.14) y Produccion_intermedia (Imagen 4.15) es que esta última, a pesar que tiene nombre, devuelve un arreglo, en vez de un token. Este operador es utilizado para la producción inicial.

```
class Categoria_lexica < Coordinador_sintactico

  def initialize nombre_de_categoria{...}
  def busqueda_de_simbolos_iniciales{...}
  def verificar_archivo{...}
  def creacion_de_enlaces{...}

  alias_method :busqueda_inicial_de_simbolos_iniciales, :busqueda_de_simbolos_iniciales
  alias_method :verificar_archivo_en_disyuncion, :verificar_archivo

end
```

Imagen 4.16: Código de implementación del operador sintáctico Categoría léxica

El operador Categoria_lexica (Imagen 4.16) se encarga de ver si coinciden el nombre que contiene la instancia de este operador con la parte nombre_de_categoria del token actual.

```
class Cadena_simple < Coordinador_sintactico

  def initialize cadena{...}
  def busqueda_de_simbolos_iniciales{...}
  def verificar_archivo{...}
  def verificar_archivo_en_disyuncion{...}

  alias_method :busqueda_inicial_de_simbolos_iniciales, :busqueda_de_simbolos_iniciales

end
```

Imagen 4.17: Código de implementación del operador sintáctico Cadena simple

El operador Cadena_simple (Imagen 4.17) se encarga de verificar la coincidencia de la cadena que contiene con la parte token del token actual. En caso de coincidir regresa el valor de la cadena, sino regresa nulo.

Capítulo 5: Pruebas de la nueva herramienta

Introducción

El proceso de verificación y validación mediante la realización de pruebas sobre VERT es un procedimiento necesario para confirmar el buen funcionamiento del sistema. Para la realización de esta etapa, se llevaron a cabo pruebas unitarias y de integración para validar cada componente y su correcta interacción con el resto del programa.

5.1. Pruebas unitarias y de integración

Una prueba unitaria es cuando se ponen a prueba los componentes de manera aislada, mientras que en una prueba de integración se verifica el funcionamiento de los componentes al interactuar con otras partes del programa principal.

Las pruebas que se aplicarán a continuación son pruebas unitarias que a su vez se unen los resultados de cada prueba para lograr las pruebas de integración

Prueba del Normalizador

Para realizar la prueba del normalizador, tenemos como entrada a la “Cadena a analizar”. La “Cadena a analizar” es el elemento a validar y procesar si cumple o no con los patrones sintácticos y gramáticos definidos, esta es la entrada del componente “Normalizador”. La ubicación de la “Cadena a analizar” se muestra en el diagrama 5.1.

Resultado esperado de la prueba: El trabajo del “Normalizador” es de transformar la “Cadena a analizar” a una secuencia de tokens donde cada token está integrado por dos partes: su categoría léxica que en este caso es nulo, ya que aún no se ha procesado léxicamente, y la otra parte es su contenido que, en este caso, cada token tendrá un caracter de la cadena original. La secuencia de caracteres del resultado de esta prueba tendrá tantos tokens como caracteres tiene la “Cadena a analizar”.

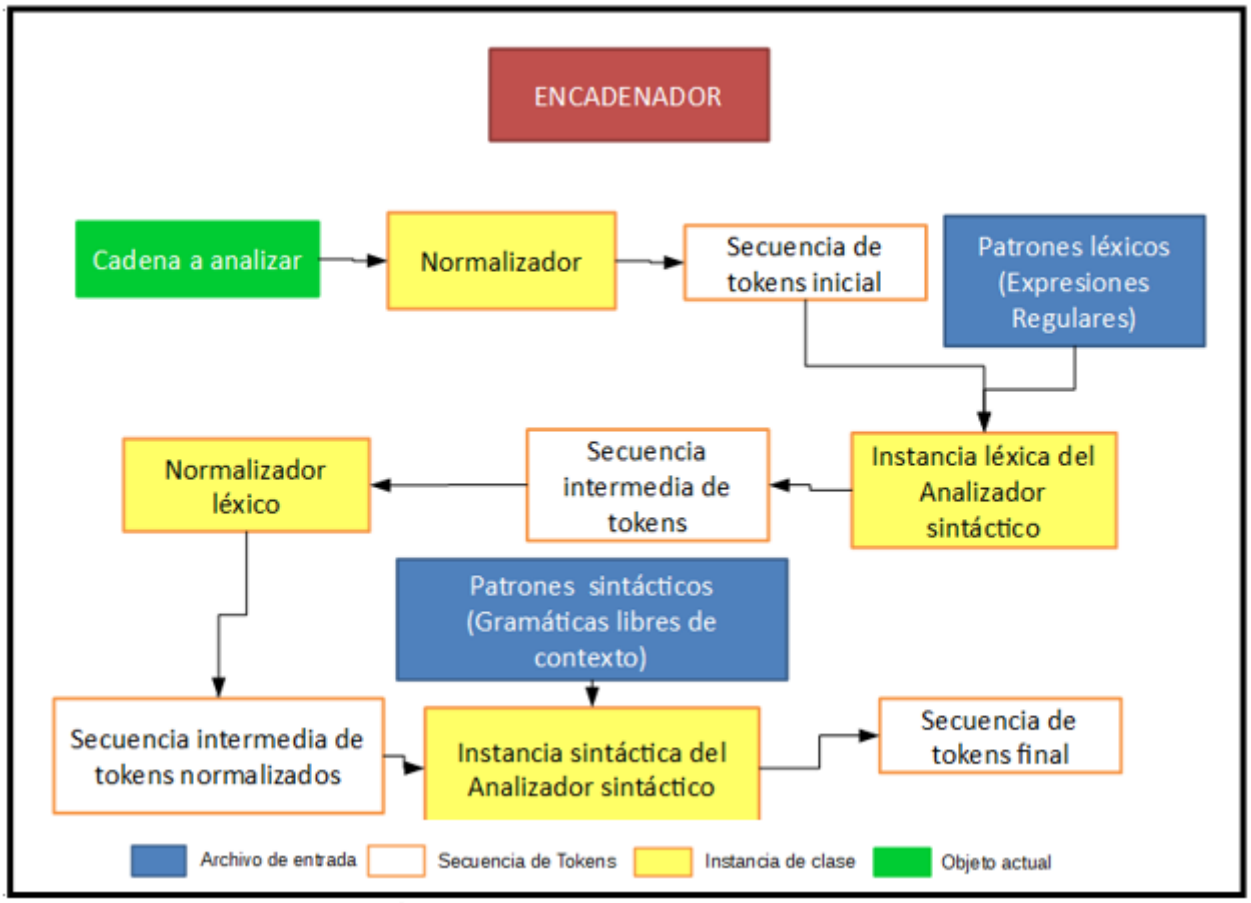


Diagrama 5.1: Cadena a analizar

A continuación se presenta la cadena a analizar que utilizaremos para realizar la prueba:

Cadena a analizar de ejemplo

```
if variable_2 =1 then
echo "hola mundo"
else
        variable_2 + variable_1
        hola = 13 *45
end
while variable_1 = 2
```

```
   variable_3 = 5*5

end

for variable_4 = 90 ; variable_4 += 1

        echo variable_4

end
```

Como resultado de ejecutar el “Normalizador” con la “Cadena a analizar” como entrada obtenemos lo siguiente:

```
[
#<struct Token nombre_de_categoria=nil, token="i">,
 #<struct Token nombre_de_categoria=nil, token="f">,
 #<struct Token nombre_de_categoria=nil, token=" ">,
 #<struct Token nombre_de_categoria=nil, token="v">,
 #<struct Token nombre_de_categoria=nil, token="a">,
 #<struct Token nombre_de_categoria=nil, token="r">,
 #<struct Token nombre_de_categoria=nil, token="i">,
 #<struct Token nombre_de_categoria=nil, token="a">,
 #<struct Token nombre_de_categoria=nil, token="b">,
 #<struct Token nombre_de_categoria=nil, token="l">,
 #<struct Token nombre_de_categoria=nil, token="e">,
 #<struct Token nombre_de_categoria=nil, token="_">,
 #<struct Token nombre_de_categoria=nil, token="2">,
 #<struct Token nombre_de_categoria=nil, token=" ">,
 #<struct Token nombre_de_categoria=nil, token="=">,
 #<struct Token nombre_de_categoria=nil, token="1">,
 #<struct Token nombre_de_categoria=nil, token=" ">,
 #<struct Token nombre_de_categoria=nil, token="t">,
 #<struct Token nombre_de_categoria=nil, token="h">,
 #<struct Token nombre_de_categoria=nil, token="e">,
 #<struct Token nombre_de_categoria=nil, token="n">,
 #<struct Token nombre_de_categoria=nil, token="\n">,
 #<struct Token nombre_de_categoria=nil, token="\n">,
 #<struct Token nombre_de_categoria=nil, token=" ">,
#<struct Token nombre_de_categoria=nil, token=" ">,
 #<struct Token nombre_de_categoria=nil, token=" ">,
#<struct Token nombre_de_categoria=nil, token="e">,
 #<struct Token nombre_de_categoria=nil, token="c">,
 #<struct Token nombre_de_categoria=nil, token="h">,
 #<struct Token nombre_de_categoria=nil, token="o">,
 #<struct Token nombre_de_categoria=nil, token=" ">,
 #<struct Token nombre_de_categoria=nil, token="\"">,
 #<struct Token nombre_de_categoria=nil, token="h">,
 #<struct Token nombre_de_categoria=nil, token="o">,
 #<struct Token nombre_de_categoria=nil, token="l">,
```

#<struct Token nombre_de_categoria=nil, token="a">,
#<struct Token nombre_de_categoria=nil, token=" ">,
#<struct Token nombre_de_categoria=nil, token="m">,
#<struct Token nombre_de_categoria=nil, token="u">,
#<struct Token nombre_de_categoria=nil, token="n">,
#<struct Token nombre_de_categoria=nil, token="d">,
#<struct Token nombre_de_categoria=nil, token="o">,
#<struct Token nombre_de_categoria=nil, token="\"">,
#<struct Token nombre_de_categoria=nil, token="\n">,
#<struct Token nombre_de_categoria=nil, token="\n">,
#<struct Token nombre_de_categoria=nil, token="e">,
#<struct Token nombre_de_categoria=nil, token="l">,
#<struct Token nombre_de_categoria=nil, token="s">,
#<struct Token nombre_de_categoria=nil, token="e">,
#<struct Token nombre_de_categoria=nil, token=" ">,
#<struct Token nombre_de_categoria=nil, token="\n">,
#<struct Token nombre_de_categoria=nil, token="\n">,
#<struct Token nombre_de_categoria=nil, token="\t">,
#<struct Token nombre_de_categoria=nil, token="v">,
#<struct Token nombre_de_categoria=nil, token="a">,
#<struct Token nombre_de_categoria=nil, token="r">,
#<struct Token nombre_de_categoria=nil, token="i">,
#<struct Token nombre_de_categoria=nil, token="a">,
#<struct Token nombre_de_categoria=nil, token="b">,
#<struct Token nombre_de_categoria=nil, token="l">,
#<struct Token nombre_de_categoria=nil, token="e">,
#<struct Token nombre_de_categoria=nil, token="_">,
#<struct Token nombre_de_categoria=nil, token="2">,
#<struct Token nombre_de_categoria=nil, token=" ">,
#<struct Token nombre_de_categoria=nil, token="+">,
#<struct Token nombre_de_categoria=nil, token=" ">,
#<struct Token nombre_de_categoria=nil, token="v">,
#<struct Token nombre_de_categoria=nil, token="a">,
#<struct Token nombre_de_categoria=nil, token="r">,
#<struct Token nombre_de_categoria=nil, token="i">,
#<struct Token nombre_de_categoria=nil, token="a">,
#<struct Token nombre_de_categoria=nil, token="b">,
#<struct Token nombre_de_categoria=nil, token="l">,
#<struct Token nombre_de_categoria=nil, token="e">,
#<struct Token nombre_de_categoria=nil, token="_">,
#<struct Token nombre_de_categoria=nil, token="1">,
#<struct Token nombre_de_categoria=nil, token="\n">,
#<struct Token nombre_de_categoria=nil, token="\t">,
#<struct Token nombre_de_categoria=nil, token="\n">,
#<struct Token nombre_de_categoria=nil, token="\t">,
#<struct Token nombre_de_categoria=nil, token="h">,
#<struct Token nombre_de_categoria=nil, token="o">,
#<struct Token nombre_de_categoria=nil, token="l">,
#<struct Token nombre_de_categoria=nil, token="a">,
#<struct Token nombre_de_categoria=nil, token=" ">,
#<struct Token nombre_de_categoria=nil, token="=">,
#<struct Token nombre_de_categoria=nil, token=" ">,

#<struct Token nombre_de_categoria=nil, token="1">,
#<struct Token nombre_de_categoria=nil, token="3">,
#<struct Token nombre_de_categoria=nil, token=" ">,
#<struct Token nombre_de_categoria=nil, token="*">,
#<struct Token nombre_de_categoria=nil, token="4">,
#<struct Token nombre_de_categoria=nil, token="5">,
#<struct Token nombre_de_categoria=nil, token="\n">,
#<struct Token nombre_de_categoria=nil, token="\n">,
#<struct Token nombre_de_categoria=nil, token="e">,
#<struct Token nombre_de_categoria=nil, token="n">,
#<struct Token nombre_de_categoria=nil, token="d">,
#<struct Token nombre_de_categoria=nil, token="\n">,
#<struct Token nombre_de_categoria=nil, token="\n">,
#<struct Token nombre_de_categoria=nil, token="\n">,
#<struct Token nombre_de_categoria=nil, token="\n">,
#<struct Token nombre_de_categoria=nil, token="\n">,
#<struct Token nombre_de_categoria=nil, token="w">,
#<struct Token nombre_de_categoria=nil, token="h">,
#<struct Token nombre_de_categoria=nil, token="i">,
#<struct Token nombre_de_categoria=nil, token="l">,
#<struct Token nombre_de_categoria=nil, token="e">,
#<struct Token nombre_de_categoria=nil, token=" ">,
#<struct Token nombre_de_categoria=nil, token="v">,
#<struct Token nombre_de_categoria=nil, token="a">,
#<struct Token nombre_de_categoria=nil, token="r">,
#<struct Token nombre_de_categoria=nil, token="i">,
#<struct Token nombre_de_categoria=nil, token="a">,
#<struct Token nombre_de_categoria=nil, token="b">,
#<struct Token nombre_de_categoria=nil, token="l">,
#<struct Token nombre_de_categoria=nil, token="e">,
#<struct Token nombre_de_categoria=nil, token="_">,
#<struct Token nombre_de_categoria=nil, token="1">,
#<struct Token nombre_de_categoria=nil, token=" ">,
#<struct Token nombre_de_categoria=nil, token="=">,
#<struct Token nombre_de_categoria=nil, token=" ">,
#<struct Token nombre_de_categoria=nil, token="2">,
#<struct Token nombre_de_categoria=nil, token="\n">,
#<struct Token nombre_de_categoria=nil, token="\n">,
#<struct Token nombre_de_categoria=nil, token=" ">,
#<struct Token nombre_de_categoria=nil, token=" ">,
#<struct Token nombre_de_categoria=nil, token=" ">,
#<struct Token nombre_de_categoria=nil, token=" ">,
#<struct Token nombre_de_categoria=nil, token="v">,
#<struct Token nombre_de_categoria=nil, token="a">,
#<struct Token nombre_de_categoria=nil, token="r">,
#<struct Token nombre_de_categoria=nil, token="i">,
#<struct Token nombre_de_categoria=nil, token="a">,
#<struct Token nombre_de_categoria=nil, token="b">,
#<struct Token nombre_de_categoria=nil, token="l">,
#<struct Token nombre_de_categoria=nil, token="e">,
#<struct Token nombre_de_categoria=nil, token="_">,
#<struct Token nombre_de_categoria=nil, token="3">,

#<struct Token nombre_de_categoria=nil, token=" ">,
#<struct Token nombre_de_categoria=nil, token="=">,
#<struct Token nombre_de_categoria=nil, token=" ">,
#<struct Token nombre_de_categoria=nil, token="5">,
#<struct Token nombre_de_categoria=nil, token="*">,
#<struct Token nombre_de_categoria=nil, token="5">,
#<struct Token nombre_de_categoria=nil, token="\n">,
#<struct Token nombre_de_categoria=nil, token="\t">,
#<struct Token nombre_de_categoria=nil, token=" ">,
#<struct Token nombre_de_categoria=nil, token=" ">,
#<struct Token nombre_de_categoria=nil, token="\n">,
#<struct Token nombre_de_categoria=nil, token="e">,
#<struct Token nombre_de_categoria=nil, token="n">,
#<struct Token nombre_de_categoria=nil, token="d">,
#<struct Token nombre_de_categoria=nil, token="\n">,
#<struct Token nombre_de_categoria=nil, token="\n">,
#<struct Token nombre_de_categoria=nil, token="\n">,
#<struct Token nombre_de_categoria=nil, token="\n">,
#<struct Token nombre_de_categoria=nil, token="\n">,
#<struct Token nombre_de_categoria=nil, token="f">,
#<struct Token nombre_de_categoria=nil, token="o">,
#<struct Token nombre_de_categoria=nil, token="r">,
#<struct Token nombre_de_categoria=nil, token=" ">,
#<struct Token nombre_de_categoria=nil, token="v">,
#<struct Token nombre_de_categoria=nil, token="a">,
#<struct Token nombre_de_categoria=nil, token="r">,
#<struct Token nombre_de_categoria=nil, token="i">,
#<struct Token nombre_de_categoria=nil, token="a">,
#<struct Token nombre_de_categoria=nil, token="b">,
#<struct Token nombre_de_categoria=nil, token="l">,
#<struct Token nombre_de_categoria=nil, token="e">,
#<struct Token nombre_de_categoria=nil, token="_">,
#<struct Token nombre_de_categoria=nil, token="4">,
#<struct Token nombre_de_categoria=nil, token=" ">,
#<struct Token nombre_de_categoria=nil, token="=">,
#<struct Token nombre_de_categoria=nil, token=" ">,
#<struct Token nombre_de_categoria=nil, token="9">,
#<struct Token nombre_de_categoria=nil, token="0">,
#<struct Token nombre_de_categoria=nil, token=" ">,
#<struct Token nombre_de_categoria=nil, token=";">,
#<struct Token nombre_de_categoria=nil, token=" ">,
#<struct Token nombre_de_categoria=nil, token="v">,
#<struct Token nombre_de_categoria=nil, token="a">,
#<struct Token nombre_de_categoria=nil, token="r">,
#<struct Token nombre_de_categoria=nil, token="i">,
#<struct Token nombre_de_categoria=nil, token="a">,
#<struct Token nombre_de_categoria=nil, token="b">,
#<struct Token nombre_de_categoria=nil, token="l">,
#<struct Token nombre_de_categoria=nil, token="e">,
#<struct Token nombre_de_categoria=nil, token="_">,
#<struct Token nombre_de_categoria=nil, token="4">,
#<struct Token nombre_de_categoria=nil, token=" ">,

#<struct Token nombre_de_categoria=nil, token="+">,
#<struct Token nombre_de_categoria=nil, token="=">,
#<struct Token nombre_de_categoria=nil, token=" ">,
#<struct Token nombre_de_categoria=nil, token="1">,
#<struct Token nombre_de_categoria=nil, token=" ">,
#<struct Token nombre_de_categoria=nil, token="\n">,
#<struct Token nombre_de_categoria=nil, token="\n">,
#<struct Token nombre_de_categoria=nil, token="\t">,
#<struct Token nombre_de_categoria=nil, token="e">,
#<struct Token nombre_de_categoria=nil, token="c">,
#<struct Token nombre_de_categoria=nil, token="h">,
#<struct Token nombre_de_categoria=nil, token="o">,
#<struct Token nombre_de_categoria=nil, token=" ">,
#<struct Token nombre_de_categoria=nil, token="v">,
#<struct Token nombre_de_categoria=nil, token="a">,
#<struct Token nombre_de_categoria=nil, token="r">,
#<struct Token nombre_de_categoria=nil, token="i">,
#<struct Token nombre_de_categoria=nil, token="a">,
#<struct Token nombre_de_categoria=nil, token="b">,
#<struct Token nombre_de_categoria=nil, token="l">,
#<struct Token nombre_de_categoria=nil, token="e">,
#<struct Token nombre_de_categoria=nil, token="_">,
#<struct Token nombre_de_categoria=nil, token="4">,
#<struct Token nombre_de_categoria=nil, token="\n">,
#<struct Token nombre_de_categoria=nil, token="\n">,
#<struct Token nombre_de_categoria=nil, token="e">,
#<struct Token nombre_de_categoria=nil, token="n">,
#<struct Token nombre_de_categoria=nil, token="d">,
#<struct Token nombre_de_categoria=nil, token="\n">,
#<struct Token nombre_de_categoria=nil, token="\n">
]

Resultado de la prueba: Obtuvimos una secuencia de caracteres con categoría léxica nula y con su contenido tenemos un caracter de la cadena original, teniendo tantos tokens en esta secuencia, como caracteres en la “Cadena a analizar”. Por lo anterior expuesto, el resultado de la prueba fue exitoso.

La secuencia anterior es la “secuencia de tokens inicial”. Su ubicación en el proceso se muestra en el diagrama 5.2.

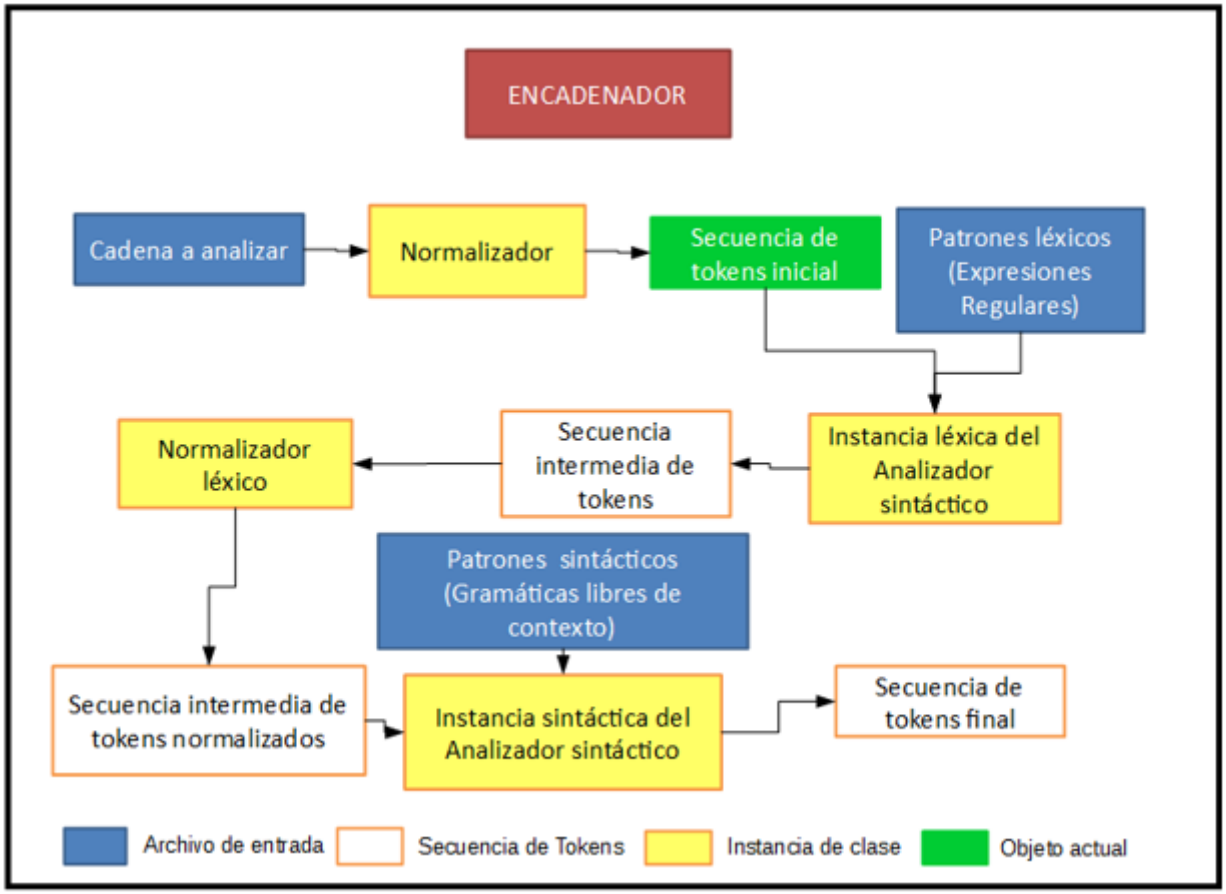


Diagrama 5.2: Secuencia de tokens inicial

Prueba de la "Instancia léxica del Analizador sintáctico"

Para esta prueba se tendrán dos entradas que son: "La secuencia de tokens inicial" y Los "Patrones léxicos".

Patrones léxicos (expresiones regulares)

Los "Patrones léxicos" son los encargados de describir el proceso léxico. Su ubicación en el proceso se muestra en el diagrama 5.3.

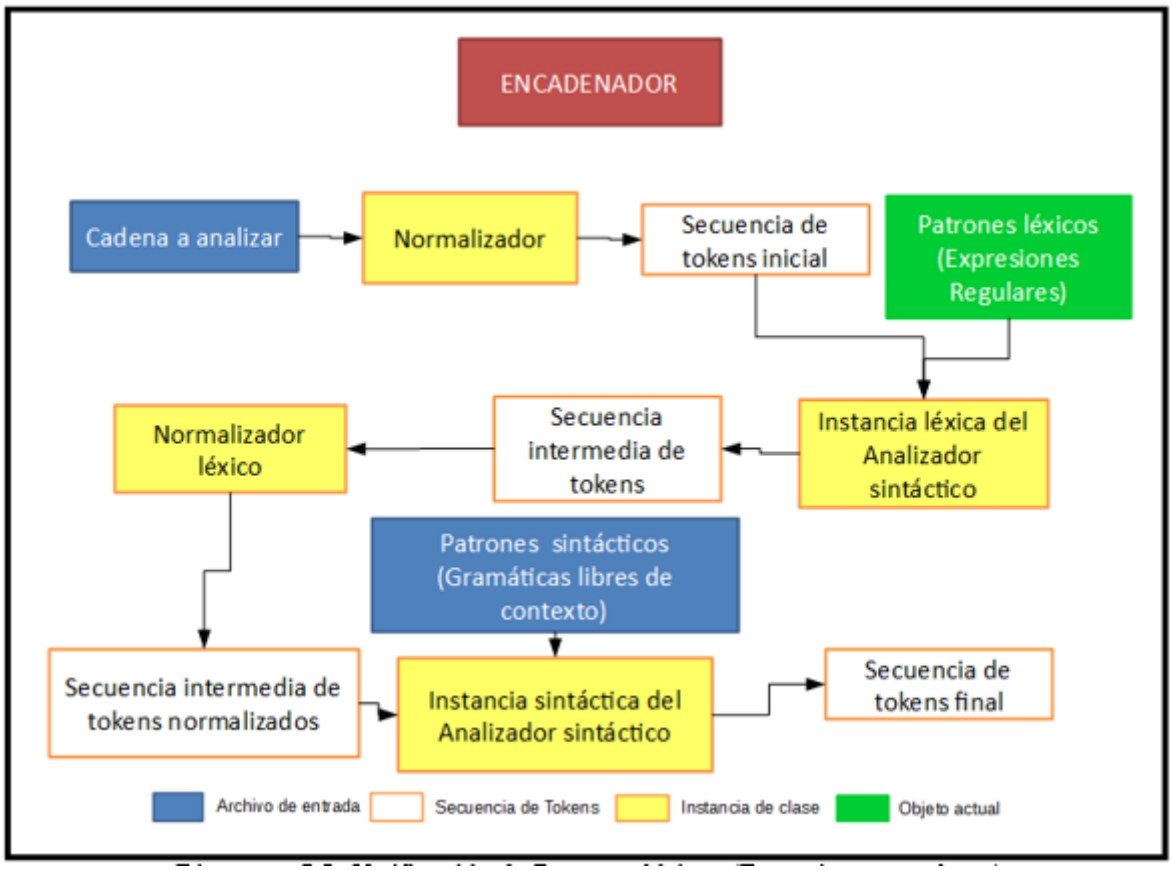


Diagrama 5.3: Patrones léxicos (Expresiones regulares)

A continuación se presenta los patrones sintácticos que se usaran de ejemplo:

Patrones léxicos de ejemplo

~inicio =

(

%identificador

|

%espacios /

|

%numero

|

%compuesto

|

%cadena

)

!

identificador =

(a | b | c | d | e | f | g | h | i | j | k | l | m | n | o | p | q | r | s | t | u | v | y | w | x | z

| A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | Y | W | X | Z

| _ | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9) +1, @conjunto_de_letras

!

espacios =

(\ | \n | \ )

!

numero =

(0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9) +1, @conjunto_de_numeros

!

cadena =

" " ^ +0, @contenido_de_cadena "

!

compuesto =

\+ @compuesto = @compuesto

!

Resultado esperado de la prueba

Se espera una secuencia de tokens donde aparecerán tokens “normales” como en la secuencia anterior y “tokens intermedios”. Los “tokens intermedios” están integrados por la categoría léxica que es el nombre del patrón léxico con el que coincidieron, y las variables con su contenido.

El resultado arrojado después de ejecutar la “Instancia léxica del Analizador sintáctico” es el siguiente:

```
[#<struct Token_intermedio
  nombre_de_categoria=:identificador,
variables={"conjunto_de_letras"=>["i", "f"]}>,
 #<struct Token_intermedio
  nombre_de_categoria=:identificador,
variables=
   {"conjunto_de_letras"=>["v", "a", "r", "i", "a", "b", "l", "e", "_", "2"]}>,
#<struct Token nombre_de_categoria=nil, token="=">,
 #<struct Token_intermedio
nombre_de_categoria=:identificador,
variables={"conjunto_de_letras"=>["1"]}>,
 #<struct Token_intermedio
  nombre_de_categoria=:identificador,
variables={"conjunto_de_letras"=>["t", "h", "e", "n"]}>,
 #<struct Token_intermedio
  nombre_de_categoria=:identificador,
variables={"conjunto_de_letras"=>["e", "c", "h", "o"]}>,
 #<struct Token_intermedio
  nombre_de_categoria=:cadena,
variables=
   {"contenido_de_cadena"=>
    ["h", "o", "l", "a", " ", "m", "u", "n", "d", "o"]}>,
 #<struct Token_intermedio
  nombre_de_categoria=:identificador,
variables={"conjunto_de_letras"=>["e", "l", "s", "e"]}>,
 #<struct Token_intermedio
  nombre_de_categoria=:identificador,
variables=
   {"conjunto_de_letras"=>["v", "a", "r", "i", "a", "b", "l", "e", "_", "2"]}>,
#<struct Token nombre_de_categoria=nil, token="+">,
 #<struct Token_intermedio
nombre_de_categoria=:identificador,
variables=
   {"conjunto_de_letras"=>["v", "a", "r", "i", "a", "b", "l", "e", "_", "1"]}>,
 #<struct Token_intermedio
  nombre_de_categoria=:identificador,
variables={"conjunto_de_letras"=>["h", "o", "l", "a"]}>,
#<struct Token nombre_de_categoria=nil, token="=">,
 #<struct Token_intermedio
nombre_de_categoria=:identificador,
variables={"conjunto_de_letras"=>["1", "3"]}>,
#<struct Token nombre_de_categoria=nil, token="*">,
 #<struct Token_intermedio
nombre_de_categoria=:identificador,
variables={"conjunto_de_letras"=>["4", "5"]}>,
 #<struct Token_intermedio
  nombre_de_categoria=:identificador,
variables={"conjunto_de_letras"=>["e", "n", "d"]}>,
```

#<struct Token_intermedio
nombre_de_categoria=:identificador,
variables={"conjunto_de_letras"=>["w", "h", "i", "l", "e"]}>,
#<struct Token_intermedio
nombre_de_categoria=:identificador,
variables=
{"conjunto_de_letras"=>["v", "a", "r", "i", "a", "b", "l", "e", "_", "1"]}>,
#<struct Token nombre_de_categoria=nil, token="=">,
#<struct Token_intermedio
nombre_de_categoria=:identificador,
variables={"conjunto_de_letras"=>["2"]}>,
#<struct Token_intermedio
nombre_de_categoria=:identificador,
variables=
{"conjunto_de_letras"=>["v", "a", "r", "i", "a", "b", "l", "e", "_", "3"]}>,
#<struct Token nombre_de_categoria=nil, token="=">,
#<struct Token_intermedio
nombre_de_categoria=:identificador,
variables={"conjunto_de_letras"=>["5"]}>,
#<struct Token nombre_de_categoria=nil, token="*">,
#<struct Token_intermedio
nombre_de_categoria=:identificador,
variables={"conjunto_de_letras"=>["5"]}>,
#<struct Token_intermedio
nombre_de_categoria=:identificador,
variables={"conjunto_de_letras"=>["e", "n", "d"]}>,
#<struct Token_intermedio
nombre_de_categoria=:identificador,
variables={"conjunto_de_letras"=>["f", "o", "r"]}>,
#<struct Token_intermedio
nombre_de_categoria=:identificador,
variables=
{"conjunto_de_letras"=>["v", "a", "r", "i", "a", "b", "l", "e", "_", "4"]}>,
#<struct Token nombre_de_categoria=nil, token="=">,
#<struct Token_intermedio
nombre_de_categoria=:identificador,
variables={"conjunto_de_letras"=>["9", "0"]}>,
#<struct Token nombre_de_categoria=nil, token=";">,
#<struct Token_intermedio
nombre_de_categoria=:identificador,
variables=
{"conjunto_de_letras"=>["v", "a", "r", "i", "a", "b", "l", "e", "_", "4"]}>,
#<struct Token_intermedio
nombre_de_categoria=:compuesto,
variables={"compuesto"=>["+", "="]}>,
#<struct Token_intermedio
nombre_de_categoria=:identificador,
variables={"conjunto_de_letras"=>["1"]}>,
#<struct Token_intermedio
nombre_de_categoria=:identificador,
variables={"conjunto_de_letras"=>["e", "c", "h", "o"]}>,
#<struct Token_intermedio

nombre_de_categoria=:identificador,
variables=
{"conjunto_de_letras"=>["v", "a", "r", "i", "a", "b", "l", "e", "_", "4"]}>,
#<struct Token_intermedio
nombre_de_categoria=:identificador,
variables={"conjunto_de_letras"=>["e", "n", "d"]}>]

Resultado de la prueba: Tal como se esperaba, se obtuvo una secuencia de tokens, donde los "tokens intermedios" tienen el nombre del patrón léxico que los reconoció y las variables con su contenido. Dado el resultado anterior, la prueba fue exitosa.

Prueba del "Normalizador léxico"

Para probar este componente necesitamos como entrada la "Secuencia intermedia de tokens" que se obtuvo en la prueba anterior. La ubicación en el proceso de la "Secuencia intermedia de tokens" se muestra en el diagrama 5.4.

Resultado esperado de la prueba: La salida esperada es una secuencia de tokens donde los "tokens intermedios" sean sustituidos por tokens "normales". Durante la transformación entre tokens, el contenido de las variables de un "token intermedio" es concatenado para obtener una cadena que será el contenido del token "normal".

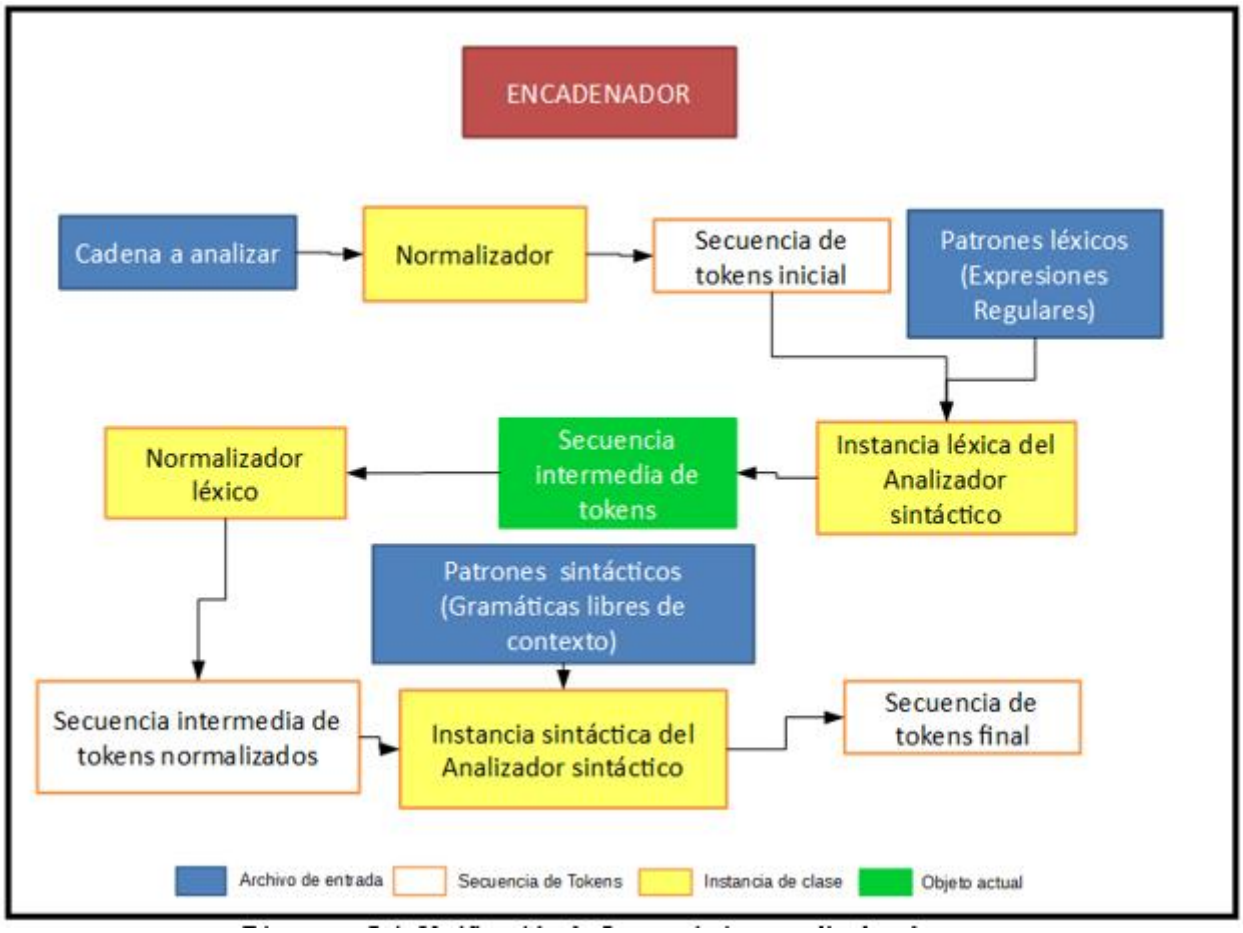


Diagrama 5.4: Verificación de Secuencia intermedia de tokens

Al ejecutar el "Normalizador léxico" teniendo como entrada la "Secuencia intermedia de tokens", obtenemos como resultado:

[
#<struct Token nombre_de_categoria=:identificador, token="if">,
#<struct Token nombre_de_categoria=:identificador, token="variable_2">,
#<struct Token nombre_de_categoria=nil, token="=">,
#<struct Token nombre_de_categoria=:identificador, token="1">,
#<struct Token nombre_de_categoria=:identificador, token="then">,
#<struct Token nombre_de_categoria=:identificador, token="echo">,
#<struct Token nombre_de_categoria=:cadena, token="hola mundo">,
#<struct Token nombre_de_categoria=:identificador, token="else">,
#<struct Token nombre_de_categoria=:identificador, token="variable_2">,
#<struct Token nombre_de_categoria=nil, token="+">,
#<struct Token nombre_de_categoria=:identificador, token="variable_1">,
#<struct Token nombre_de_categoria=:identificador, token="hola">,
#<struct Token nombre_de_categoria=nil, token="=">,
#<struct Token nombre_de_categoria=:identificador, token="13">,
#<struct Token nombre_de_categoria=nil, token="*">,
#<struct Token nombre_de_categoria=:identificador, token="45">,
#<struct Token nombre_de_categoria=:identificador, token="end">,

```
#<struct Token nombre_de_categoria=:identificador, token="while">,
#<struct Token nombre_de_categoria=:identificador, token="variable_1">,
#<struct Token nombre_de_categoria=nil, token="=">,
#<struct Token nombre_de_categoria=:identificador, token="2">,
#<struct Token nombre_de_categoria=:identificador, token="variable_3">,
#<struct Token nombre_de_categoria=nil, token="=">,
#<struct Token nombre_de_categoria=:identificador, token="5">,
#<struct Token nombre_de_categoria=nil, token="*">,
#<struct Token nombre_de_categoria=:identificador, token="5">,
#<struct Token nombre_de_categoria=:identificador, token="end">,
#<struct Token nombre_de_categoria=:identificador, token="for">,
#<struct Token nombre_de_categoria=:identificador, token="variable_4">,
#<struct Token nombre_de_categoria=nil, token="=">,
#<struct Token nombre_de_categoria=:identificador, token="90">,
#<struct Token nombre_de_categoria=nil, token=";">,
#<struct Token nombre_de_categoria=:identificador, token="variable_4">,
#<struct Token nombre_de_categoria=:compuesto, token="+=">,
#<struct Token nombre_de_categoria=:identificador, token="1">,
#<struct Token nombre_de_categoria=:identificador, token="echo">,
#<struct Token nombre_de_categoria=:identificador, token="variable_4">,
#<struct Token nombre_de_categoria=:identificador, token="end">
]
```

Resultado de la prueba: Como se esperaba, se reemplazaron los “tokens intermedios” por tokens “normales”. El contenido de estos últimos corresponde a la concatenación del contenido de las variables de los “tokens intermedios” que reemplazaron, mientras que el nombre de la categoría léxica se conservó. Con relación al resultado de la prueba y lo esperado por ella, se concluye que fue exitosa.

Prueba de la “Instancia sintáctica del analizador sintáctico”

Para llevar a cabo esta prueba se ejecutará la “Instancia sintáctica del analizador sintáctico” teniendo como entrada:

-La “Secuencia intermedia de tokens normalizados” que es el resultado de la prueba anterior. Su ubicación en el proceso se muestra en el diagrama 5.5.

-Los “Patrones sintácticos (Gramáticas libres de contexto)”. Su ubicación en el proceso se muestra en el diagrama 5.6.

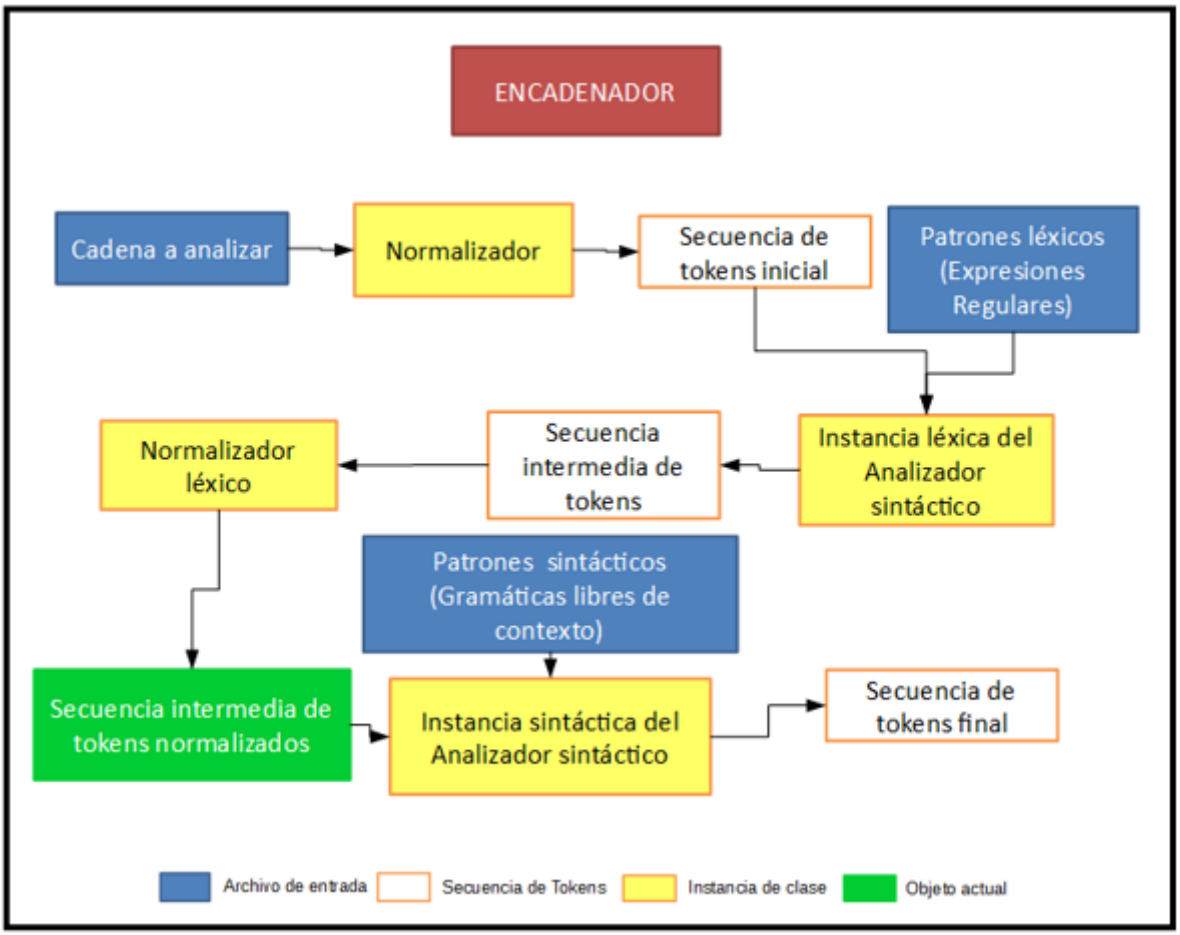


Diagrama 5.5: Secuencia intermedia de tokens normalizados

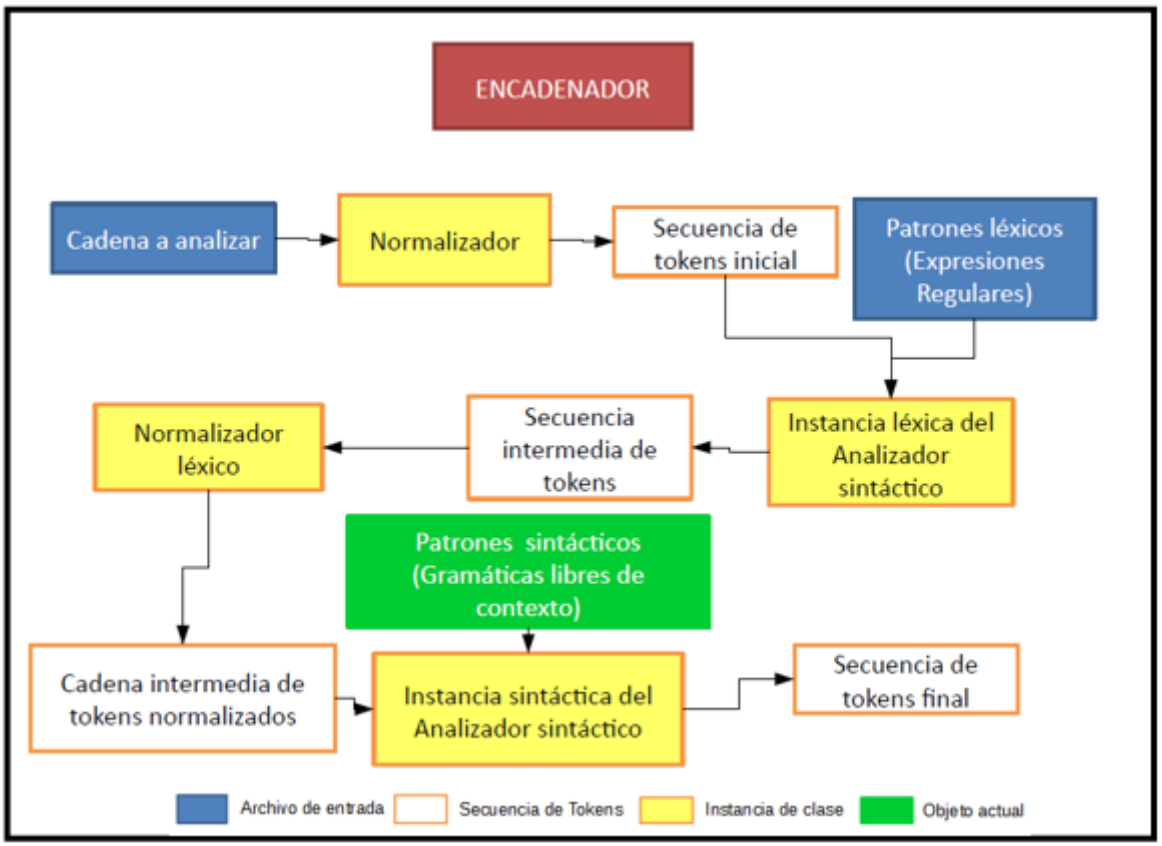


Diagrama 5.6: Patrones sintácticos (Gramática libre de contexto)

Los "Patrones sintácticos (Gramática libre de contexto)" que se usará para esta prueba son los siguientes:

Gramática de prueba

~inicial =

(

%for

|

%while

|

%if

) +1,

!

```
for =
        for  %comparacion @comparacion ; %operacion @suma_de_variable
                %operacion +1,  @contenido_de_for
        end
!
if =
if %comparacion @comparacion then
    %operacion +1,    @contenido_de_if
else
          %operacion +1,      @contenido_de_else
end
!
while =
while %comparacion  @comparacion
             %operacion +1,    @contenido_de_while
        end
!
comparacion =
        ( &numero | &identificador )  @valor_1 ( = | < | > ) @simbolo  ( &numero |
&identificador )  @valor_2
!
operacion =
        (
                echo ( &cadena | &identificador )
                |
                %suma
```

```
                |
                %multiplicacion
                |
                %asignacion_con_suma
                |
                %asignacion
        ) @operacion
!
suma =
        ( &numero | &identificador ) @valor_1  \+ ( &numero | &identificador )  @valor_2
!
multiplicacion =
        ( &numero | &identificador )  @valor_1 * ( &numero | &identificador )   @valor_2
!
asignacion =
        &identificador @valor_1 = ( %suma | %multiplicacion | &numero | &identificador )
@valor_2
!
asignacion_con_suma =
        &identificador  @valor_1 \+= ( %suma | %multiplicacion | &numero | &identificador )
@valor_2
!
```

Resultado esperado de la prueba: El resultante de esta prueba debe ser una secuencia de tokens entre los que aparecerá “tokens intermedios” conteniendo a su vez “tokens intermedios” que comprenden la estructura reconocida por los “Patrones sintácticos (Gramática libre de contexto)”.

Al ejecutar la “Instancia sintáctica del Analizador sintáctico”, teniendo como entradas la “Secuencia intermedia de tokens normalizados” y los “Patrones sintácticos (Gramática libre de contexto)”, obtenemos el siguiente resultado:

```
[#<struct Token_intermedio
  nombre_de_categoria=:if,
variables=
  {"comparacion"=>
    [#<struct Token_intermedio
      nombre_de_categoria=:comparacion,
variables=
      {"valor_1"=>["variable_2"], "simbolo"=>["="], "valor_2"=>["1"]}>],
   "contenido_de_if"=>
    [#<struct Token_intermedio
      nombre_de_categoria=:operacion,
variables={"operacion"=>["echo", "hola mundo"]}>],
   "contenido_de_else"=>
    [#<struct Token_intermedio
      nombre_de_categoria=:operacion,
variables=
      {"operacion"=>
        [#<struct Token_intermedio
          nombre_de_categoria=:suma,
variables=
          {"valor_1"=>["variable_2"], "valor_2"=>["variable_1"]}>]}>,
     #<struct Token_intermedio
      nombre_de_categoria=:operacion,
variables=
      {"operacion"=>
        [#<struct Token_intermedio
          nombre_de_categoria=:asignacion,
variables=
          {"valor_1"=>["hola"],
           "valor_2"=>
            [#<struct Token_intermedio
              nombre_de_categoria=:multiplicacion,
variables={"valor_1"=>["13"], "valor_2"=>["45"]}>]}>]}>]}>,
 #<struct Token_intermedio
  nombre_de_categoria=:while,
variables=
  {"comparacion"=>
    [#<struct Token_intermedio
```

**nombre_de_categoria=:comparacion,
variables=
{"valor_1"=>["variable_1"], "simbolo"=>["="], "valor_2"=>["2"]}>],
"contenido_de_while"=>
[#<struct Token_intermedio
nombre_de_categoria=:operacion,
variables=
{"operacion"=>
[#<struct Token_intermedio
nombre_de_categoria=:asignacion,
variables=
{"valor_1"=>["variable_3"],
"valor_2"=>
[#<struct Token_intermedio
nombre_de_categoria=:multiplicacion,
variables={"valor_1"=>["5"], "valor_2"=>["5"]}>]}>]}>]}>,
#<struct Token_intermedio
nombre_de_categoria=:for,
variables=
{"comparacion"=>
[#<struct Token_intermedio
nombre_de_categoria=:comparacion,
variables=
{"valor_1"=>["variable_4"], "simbolo"=>["="], "valor_2"=>["90"]}>],
"suma_de_variable"=>
[#<struct Token_intermedio
nombre_de_categoria=:operacion,
variables=
{"operacion"=>
[#<struct Token_intermedio
nombre_de_categoria=:asignacion_con_suma,
variables={"valor_1"=>["variable_4"], "valor_2"=>["1"]}>]}>],
"contenido_de_for"=>
[#<struct Token_intermedio
nombre_de_categoria=:operacion,
variables={"operacion"=>["echo", "variable_4"]}>]}>]**

Resultado de la prueba: Se obtuvo una secuencia de “tokens intermedios” que contienen la estructura sintáctica reconocida por los “Patrones sintácticos (Gramática libre de contexto)”. Esta secuencia de tokens es la “Secuencia final de tokens”. Esta secuencia contiene el resultado final que devolverá VERT, la ubicación de esta secuencia se muestra al final del proceso en el diagrama 5.7.

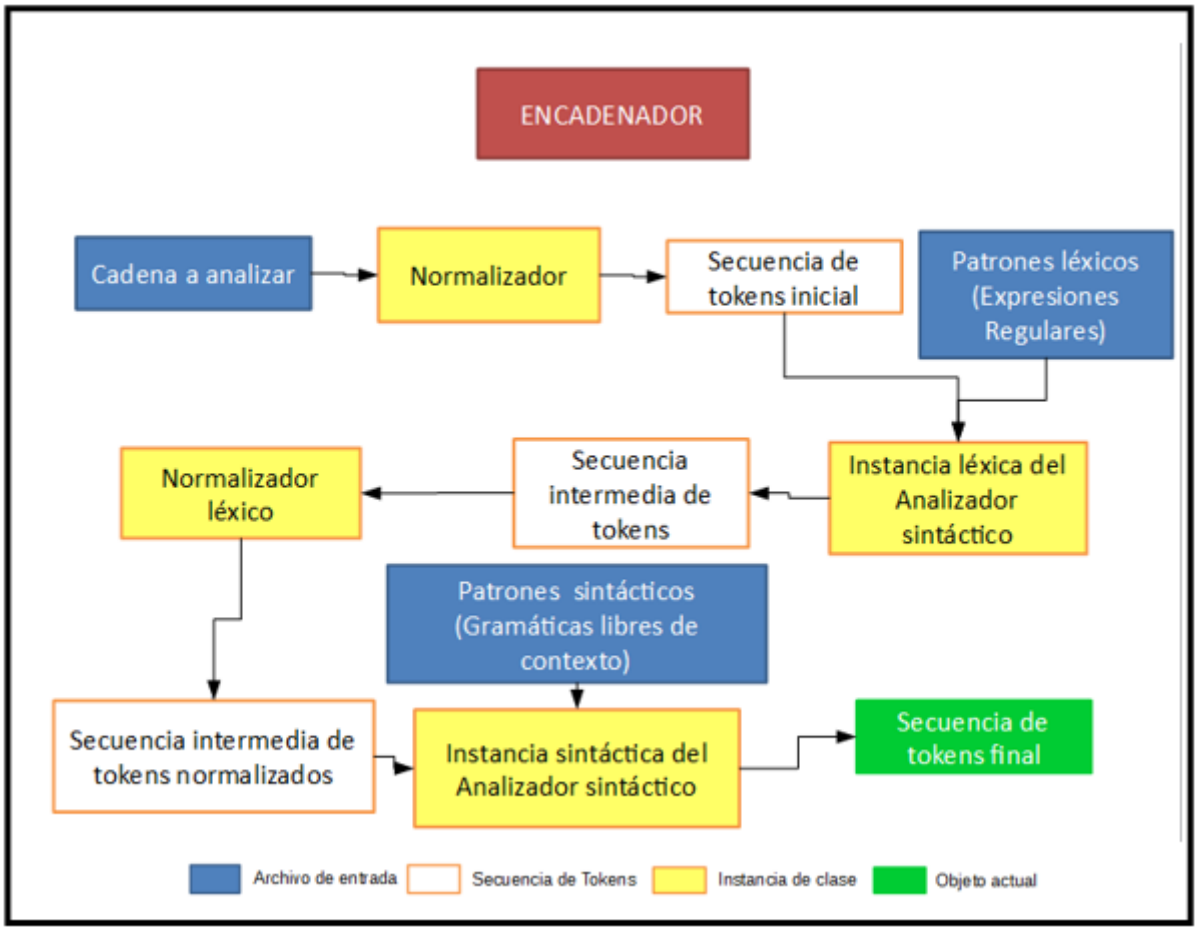


Diagrama 5.7: Secuencia de tokens final

Conclusiones de las pruebas

Cada una de las pruebas unitarias y de integración resultaron exitosas y en conjunto se obtuvo el resultado esperado, es decir, teniendo como entrada la “Cadena a analizar”, los “Patrones léxicos (expresiones regulares)” y los “Patrones sintácticos (Gramática libre de contexto)”, se obtuvo una “Secuencia final de tokens” que es el resultado de los análisis léxico y sintáctico definidos. Por lo anterior se puede concluir que las pruebas llevadas a cabo dieron resultados satisfactorios.

Capítulo 6: Resultados

Las pruebas aplicadas a VERT se cumplieron satisfactoriamente. Cada uno de los componentes, que integran a VERT, fueron probados individualmente y de manera conjunta para comprobar que funcionaban correctamente al integrarse con los otros componentes.

Se obtuvo como resultado del presente trabajo, una nueva herramienta la cual es un generador de analizadores léxicos y sintácticos, el cual cuenta con las siguientes características:

-Rápido desarrollo de los analizadores léxicos y sintácticos, ya que para efectuar una modificación en ellos sólo basta con cambiar la definición de la parte deseada, ya sea léxica o sintáctica, y ejecutar directamente el procesamiento sin tener que pasar por una etapa de compilación, debido a que VERT es una herramienta interpretada, no compilada.

-No genera errores sintácticos o léxicos, ya que para VERT entrega como resultado una secuencia de tokens que dependen de la definición léxica y sintáctica. Si una parte de la cadena que se dio como entrada a VERT coincide con los patrones léxicos y sintácticos definidos por el usuario, entonces los agrupa en un “token intermedio” que contiene como categoría léxica el nombre de la última producción que cumplió con el patrón y, por otra parte, las variables y su contenido. El resto de las partes que no sean identificadas son devueltas como tokens “normales”.

-El lenguaje utilizado para la definición de la parte léxica y sintáctica, es el mismo.

-En los archivos de definición de las partes léxica y sintáctica, no se incluye código del lenguaje en el que fue implementado VERT, en este caso Ruby. Se puede observar el contraste de este punto en los códigos de ejemplo del capítulo 2 tanto para Lex, Yacc y Javacc.

-Los análisis léxico y sintáctico se hacen con VERT, no depende de otro programa como Lex o Yacc para realizar alguno de estos análisis.

Capítulo 7: Conclusiones

Resultados esperados planteados al inicio del trabajo:

-Construcción de un generador de analizadores léxicos y sintácticos.

-Independencia entre el lenguaje de definición léxica y sintáctica, y el lenguaje de programación en el que se encuentra implementada la nueva herramienta.

-El lenguaje que se utiliza en la definición léxica es el mismo que en la definición sintáctica.

Al comparar los resultados esperados y los resultados obtenidos mostrados en el capítulo anterior, se puede concluir que los objetivos de la presente tesis se cumplieron satisfactoriamente.

Conclusiones grupales

Una vez encontrada la problemática presente y la necesidad de solventarla, el diseño, implementación y pruebas de una herramienta de esta complejidad fue un reto bastante desafiante, ya que la complejidad que representa los análisis léxico y sintáctico puso a prueba los conocimientos y habilidades que adquirimos a lo largo de nuestra formación profesional.

Durante el desarrollo de esta tesis nos enfrentamos principalmente a tres obstáculos: la distribución del trabajo, las reuniones para llevar un control de los avances y, por último, el consenso de ideas.

Fue complicado reunirnos de manera física, debido a nuestras diversas actividades, entre ellas nuestras actividades laborales, por lo que tuvimos que hacer uso de herramientas tecnológicas como teleconferencias y documentos de modificación colaborativa.

El consenso de nuestras diferentes ideas y puntos de vista ha sido un aprendizaje muy importante, pues en la vida profesional nos enfrentamos a este tipo de situaciones. Dos herramientas que aprendimos durante este proceso de trabajo en equipo fueron la habilidad de escuchar los puntos de vista de los otros y reflexionar de manera objetiva sobre los nuestros.

La estructuración de la tesis fue un obstáculo que pudimos resolver gracias a la ayuda de nuestro director de tesis y a nuestros sinodales, a los cuales les agradecemos enormemente su atención.

Mesografía y bibliografía

-Kenneth C. Louden (1997). Compiler Construction: Principles and Practice.

-Alfred V. Aho, Monica S. Lam, Ravi Sethi y Jeffrey D. Ullman (2006). Compilers: Principles, Techniques, and Tools

-Syntax Analysis Or Parsing, Lecture 07-08, Iffat Anjum, BRAC University

-PITTMAN, Thomas, PETERS, James, The Art of Compiler Design; Theory and Practice, Englewood cliffs New Jersey USA, Prentice Hall, 1992

-TREMBLAY, Jean-Paul, SORENSON, Paul G., The Theory and practice of compiler writing, U.S.A., Mc. Graw-Hill, 1985

-Flex & Bison, Jonh Levine, O'REILLY, USA, 2009

-AHO, A. V., SETHI, Ravi, ULLMAN, J.D., Compiladores. Principios, técnicas y herramientas., México, Addison-Wesley Iberoamericana, 2000

-TREMBLAY, Jean-Paul, SORENSON, Paul G., The Theory and practice of compiler writing, U.S.A., Mc. Graw-Hill, 1985

-SCOTT, Michael L., Programming Language Pragmatics, U.S.A., Morgan Kaufmann, 2000

-http://webdiis.unizar.es/~ezpeleta/lib/exe/fetch.php?media=misdatos:compi:2bis.introfLex.pdf, 06/01/2016, 6:24 PM

-http://www.infor.uva.es/~mluisa/talf/docs/labo/L8.pdf, 01/03/2016, 11:17 PM

-https://www.cs.rochester.edu/~nelson/courses/csc_173/grammars/cfg.html, 30/04/2016, 3:12 PM