Facultad de Ciencias de la Computación

Sistema para verificar si una Gramática Libre de Contexto es LL(1)

Tesis que presenta:

Gabriel Oscar Reyes Severiano

Director de la Tesis:

M.C. Yolanda Moyao Martínez

Resumen

Un compilador se encarga fundamentalmente de traducir, es decir, un programa que nos permite pasar información de un lenguaje a otro. Para su correcto funcionamiento el proceso de compilado se divide en distintas fases y una de ellas es el análisis sintáctico, la sintaxis es conformada por un conjunto de reglas necesarias para construir correctamente expresiones o sentencias utilizadas para el funcionamiento de una computadora, o sea que el analizador sintáctico reconoce si una o varias cadenas están estructuradas de forma correcta dentro de un conjunto de análisis posibles, este conjunto esta definido por una gramática, si la estructura gramatical no es reconocida se lanza un error, además el analizador sintáctico convierte el texto de entrada en otras estructuras (comúnmente arboles).

Los analizadores sintácticos que se basan en gramáticas LL(1) suelen utilizar como técnica predecir que regla se debe aplicar para la variable que hay que derivar, analizan de izquierda a derecha utilizando la derivación del no terminal que aparece mas a la izquierda y únicamente se recibe un token a la vez para así saber el tipo de producción gramatical a seguir.

En este proyecto se pretende desarrollar una herramienta didáctica la cual permita verificar si una gramática es o no LL(1) para ello se implementarán los algoritmos utilizados en el cálculo de los conjuntos primeros (first), siguientes (follow) y Predicttivos con los que finalmente se podrá verificar la condición LL(1) .

Índice general

Índice de figuras

Introducción

El desarrollo de compiladores e incluso la mejora de estos incentiva a desarrollar herramientas que sirvan como catapulta hacia el mejoramiento de métodos actuales para la resolución de problemas cotidianos e incluso no tan cotidianos. Los compiladores marcan el nivel tanto en software como en arquitectura de hardware, y para el desarrollo de un buen compilador es necesario esquematizar en distintas capas la maquinaria que contendrá para asegurar su funcionamiento adecuado. Dentro de las fases de estructuración de un compilador se incluye la etapa de análisis, fundamental para la lectura de texto fuente y análisis sintáctico así como la interpretación mediante un árbol que será de utilidad en la siguiente fase del compilado que consiste en la revisión semántica. Todo lenguaje de programación tiene reglas que describen la estructura sintáctica de programas bien formados. Se puede describir la sintaxis de las construcciones de los lenguajes de programación por medio de gramáticas de contexto libre o notación BNF (Backus-Naur Form).

- Las gramáticas ofrecen ventajas significativas a los diseñadores de lenguajes y a los desarrolladores de compiladores.
- Las gramáticas son especificaciones sintácticas y precisas de lenguajes de programación.
- A partir de una gramática se puede generar automáticamente un analizador sintáctico.
- Una gramática proporciona una estructura a un lenguaje de programación, siendo más fácil generar código y detectar errores.
- Es más fácil ampliar/modificar el lenguaje si está descrito con una gramática.

Como se puede observar cuando hablamos de compiladores de forma intrínseca nos referimos a Gramáticas Libres de Contexto, pues ellas conllevan a la generación y/o revisión de sintaxis especifica para determinado lenguaje de programación. La enseñanza de compiladores básicamente utiliza gramáticas del tipo LL1 para ilustrar el análisis sintáctico descendente usando herramientas de predicción que nos puede ahorrar el trabajo recursivo de prueba y error en cada regla, específicamente leyendo un carácter de entrada sabremos que regla de derivación de la gramática utilizar.

Motivación

Actualmente se cuenta con algunas herramientas que nos ayudan a comprender el proceso de análisis de sintaxis en cierta manera pero no siempre se muestra cómo es que se llevó a cabo la búsqueda entre las gramáticas, más en concreto, no se proyecta el cálculo de primeros (first) , siguientes (follow) y predictivos (predict) dentro de las gramáticas; la correcta comprensión de cálculo de primeros y siguientes mediante gramáticas LL es el motor principal para un buen planteamiento de análisis sintáctico particularmente esta comprensión es necesaria en la materia de compiladores, que se imparte regularmente en instituciones de nivel superior dedicadas a la enseñanza de computación y ciencias de la computación.

Objetivos

Objetivo General.

Se pretende crear un material de apoyo didáctico para los profesores y estudiosos en el ramo de compiladores para lograr estudiar así como comprender los mecanismos de predicción utilizados para identificar si una gramática libre de contexto es del tipo LL1.

Objetivos Específicos:

- Implementar algoritmo para eliminación de recursividad por la izquierda.
- Implementar algoritmo para factorización de gramáticas por la izquierda.
- Implementar el algoritmo para obtener el conjunto de PRIMEROS.
- Implementar el algoritmo para obtener el conjunto de SIGUIENTES.
- Implementar el algoritmo para obtener el conjunto de PREDICTIVOS.
- Probar la condición LL(1).

Capítulo 1

Marco Teórico

La base teórica es sin duda la antesala en una investigación pues nutre con conocimientos específicos hacia el tema a tratar, en el actual capítulo se hace énfasis en definiciones básicas e importantes, requeridas en la fase de implementación.

1.1. Compiladores

Los lenguajes de programación son notaciones que describen los cálculos de las maquinas. Nuestra percepción del mundo real depende de los lenguajes de programación, ya que todo software que se ejecuta en las computadoras fue escrito en determinado lenguaje de programación. Pero antes de poder ejecutar un programa, primero debe traducirse a un formato legible para la computadora y poder realizar la ejecución de las instrucciones. Los sistemas de software que se encargan de esta traducción se llaman compiladores. El estudio de la escritura de compiladores se relaciona con los lenguajes de programación, la arquitectura de las máquinas, la teoría de lenguajes, los algoritmos y la ingeniería de software.

1.1.1. Procesadores de lenguaje

Dicho en forma simple, un compilador es un programa que puede leer un programa en un lenguaje (el lenguaje fuente) y traducirlo en un programa equivalente en otro lenguaje (el lenguaje destino); vea la figura 1.1. Una función importante del compilador es reportar cualquier error en el programa fuente que detecte durante el proceso de traducción.

Si el programa de destino es un programa ejecutable en lenguaje máquina, entonces el usuario puede ejecutarlo para procesar las entradas y producir salidas (resultados); vea la figura .

Un *intérprete* es otro tipo común de procesador de lenguaje. En vez de producto un programa destino como una traducción, el interprete nos da la apariencia de ejecutar directamente las operaciones especificadas en el programa de origen (fuente) con las entradas proporcionadas por el usuario, como se muestra en la figura .

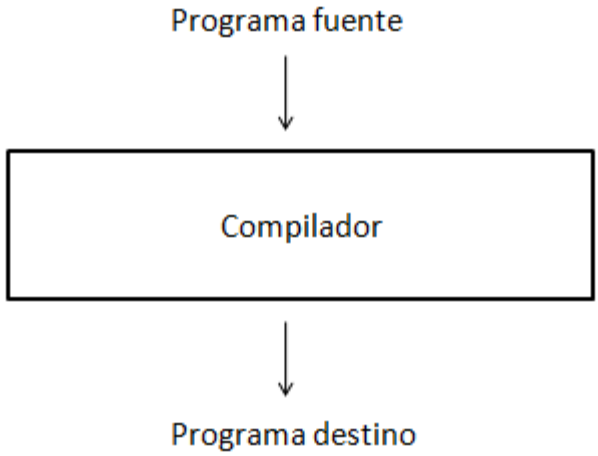


Figura 1.1: Un compilador

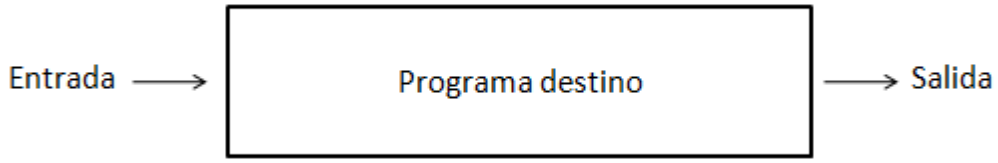


Figura 1.2: Ejecución del programa de destino

Figura 1.3: Un intérprete

1.2. Gramáticas libres de contexto

Los lenguajes libres de contexto (abreviado LLC) son importantes tanto del punto de vista teórico, como desde el punto de vista práctico, ya que casi todos los lenguajes de programación están basados en los LLC. A partir de los años 70´s, con lenguajes como Pascal, se hizo común la práctica de formalizar la sintaxis de los lenguajes de programación usando herramientas basadas en Gramáticas Libres de Contexto, que representan a los LLC. Por otra parte, el análisis automático de los LLC es computacionalmente mucho más eficiente que otras clases de lenguajes más generales.

1.2.1. Reglas

Una regla es una expresión de la forma $\alpha \rightarrow \beta$, en donde tanto α como β son cadenas de símbolos en donde pueden aparecer tanto elementos del alfabeto Σ (llamados constantes o terminales), así como nuevos símbolos llamados variables (no terminales).

1.2.2. Gramática

Una gramática es básicamente un conjunto de reglas que permite especificar un lenguaje o lengua, es decir, es el conjunto de reglas capaces de generar todas las posibles combinatorias de ese lenguaje.

Consideremos por ejemplo, la siguiente gramática para producir un pequeño subconjunto del idioma español:

1. $< frase > \rightarrow < sujeto >< predicado >$
2. $< sujeto > \rightarrow < articulo >< sustantivo >$
3. $< articulo > \rightarrow el \mid la$
4. $< sustantivo > \rightarrow perro \mid luna$
5. $< predicado > \rightarrow < verbo >$
6. $< verbo > \rightarrow brilla \mid corre$

Una gramática parte de una variable, llamada símbolo inicial, y se aplican repetidamente las reglas gramaticales, hasta que ya no halla variables en la palabra. En ese momento se dice que la palabra resultante es generada por la gramática, o en forma equivalente, que la palabra resultante es parte del lenguaje de esa gramática.

Las Gramáticas Libres de Contexto (GLC), o de tipo 2 en la jerarquía de Chomsky están constituidas por reglas de la forma $X \rightarrow \alpha$, donde X es una variable y α es una cadena que puede contener variables y constantes. Estas gramáticas producen los lenguajes libres de contexto (abreviado LLC).

1.2.3. Formalización de las GLC

Definición: Una gramática libre de contexto es un cuádruplo (V, Σ, R, S) en donde:

- V es un *alfabeto de variables*, también llamadas no-terminales.
- Σ es un *alfabeto de constantes*, también llamadas terminales. Suponemos que V y Σ son disjuntos, esto es, $V \cap \Sigma = \emptyset$.
- R , es el conjunto de *reglas de producción*.
- S, el *símbolo inicial*, es un elemento de V.

Definición: Una cadena $\alpha \epsilon (V \cup \Sigma)^*$ *es derivable* a partir de una gramática (V, Σ, R, S) si hay al menos una secuencia de pasos de derivación que la produce a partir del símbolo inicial S, esto es: $S \Rightarrow ... \Rightarrow \alpha$

Definición: El lenguaje $L(G)$ generado por una gramática (V, Σ, R, S) es el conjunto de palabras hechas exclusivamente de constantes, que son derivables a partir del símbolo inicial:

$$L = \{w \epsilon \Sigma^* | \, S \Rightarrow ... \Rightarrow w\}$$

1.2.4. Diseño de GLC

El problema del diseño de GLC consiste en proponer, dado un lenguaje L, una GLC G tal que su lenguaje generado es exactamente L. Decimos que una GLC G es *correcta* con respecto al lenguaje dado L cuando el lenguaje generado por G no contiene palabras que estén fuera de L, es decir, $\mathcal{L}(G) \subseteq L$, donde $\mathcal{L}(G)$ denota el lenguaje generado por G. Similarmente, decimos que G es completa cuando G es capaz de generar al menos las palabras de L, es decir, $L \subseteq \mathcal{L}(G)$. Al diseñar gramáticas, es posible cometer dos clases de errores:

1. Que "sobren palabras", esto es, que la gramática genere algunas palabras que no deberían generar. En este caso, la gramática sería *incorrecta*.
2. Que "falten palabras", esto es, que haya palabras en el lenguaje considerado para las que no hay ninguna derivación. En este caso, la gramática sería *incompleta*.

1.2.5. Derivaciones

La construcción de un árbol de análisis sintáctico puede hacerse precisa si tomamos una vista derivacional, en la cual las producciones se tratan como reglas de rescritura. Empezando con el símbolo inicial, cada paso de rescritura sustituye a un no terminal por el cuerpo de una de sus producciones a este proceso se le conoce como "paso de derivación", y se denota usando una flecha gruesa " $\Rightarrow$ ".

Ejemplo: Tenemos una gramática que permite generar expresiones aritméticas con sumas y multiplicaciones de enteros con las reglas siguientes:

1. $E \rightarrow E + T$
2. $E \rightarrow T$
3. $T \rightarrow T * F$
4. $T \rightarrow F$
5. $F \rightarrow CF$
6. $F \rightarrow C$
7. $C \rightarrow 0 \,|\, 1 \,|\, 2 \,|\, 3 \,|\, 4 \,|\, 5 \,|\, 6 \,|\, 7 \,|\, 8 \,|\, 9$

El símbolo inicial aquí es E, las constantes son $+, *$ y las cifras 0...9; E, T, F y C son variables.

Con esta gramática se puede generar, por ejemplo, la expresión $25 + 3 * 12$ de la siguiente manera:

EXPRESION	JUSTIFICACION
E	Símbolo incial, inicia derivación
$\Rightarrow E + T$	Aplicación 1a. regla
$\Rightarrow T + T$	2a. regla, sobre la E
$\Rightarrow F + T$	4a. regla, sobre la T izquierda
$\Rightarrow CF + T$	5a. regla, sobre F
$\Rightarrow 2F + T$	7a. regla
$\Rightarrow 2C + T$	6a. regla
$\Rightarrow 25 + T$	7a. regla
$\Rightarrow 25 + T * F$	3a. regla
$\Rightarrow 25 + F * F$	4a. regla
$\Rightarrow 25 + C * F$	6a. regla, sobre la F izquierda
$\Rightarrow 25 + 3 * F$	7a. regla
$\Rightarrow 25 + 3 * CF$	5a. regla
$\Rightarrow 25 + 3 * 1F$	7a. regla
$\Rightarrow 25 + 3 * 1C$	6a. regla
$\Rightarrow 25 + 3 * 12$	7a. regla

1.2.6. Derivacion por la izquierda y por la derecha

En una gramática no ambigua G, a una palabra $w \epsilon \mathcal{L}(G)$ corresponde un sólo árbol de derivación; sin embargo, puede haber varias derivaciones para obtener w a partir del símbolo inicial, $S \Rightarrow ... \Rightarrow w$. Una manera de hacer única la manera de derivar una palabra consiste en restringir la elección del símbolo que se va a *"expandir"* en curso de la derivación. Por ejemplo, si tenemos en cierto momento de la derivación la palabra

$(S())(S)$, en el paso siguiente podemos aplicar alguna regla de la gramática ya sea a la primera o a la segunda de las S. En cambio, si nos restringimos a aplicar las reglas solo al no terminal que se encuentre más a la izquierda en la palabra, entonces habrá una sola opción posible.

- Derivación más a la izquierda *(leftmost derivation):* $\Rightarrow_{lm}$ siempre reemplaza la variable más a la izquierda por uno de los cuerpos de sus producciones.
- Derivación más a la derecha *(rightmost derivation):* $\Rightarrow_{rm}$ siempre reemplaza la variable más a la derecha por uno de los cuerpos de sus producciones.

Ejemplo: Para la gramática no ambigua con reglas $S \Rightarrow AB$, $A \Rightarrow a$, $B \Rightarrow b$, la palabra ab se produce con la derivación izquierda:

$$S \Rightarrow AB \Rightarrow aB \Rightarrow ab$$

mientras que también se puede producir con la derivación derecha:

$$S \Rightarrow AB \Rightarrow Ab \Rightarrow ab$$

1.3. Arboles de análisis sintáctico y derivaciones

Un árbol de derivación permite mostrar gráficamente cómo se puede derivar cualquier cadena de un lenguaje a partir del símbolo distinguido de una gramática que genera ese lenguaje. Un árbol es un conjunto de puntos, llamados nodos, unidos por líneas, llamadas arcos. Un arco conecta dos nodos distintos. Para ser un árbol un conjunto de nodos y arcos debe satisfacer ciertas propiedades:

- Hay un único nodo distinguido, llamado raíz (se dibuja en la parte superior) que no tiene arcos incidentes.
- Todo nodo c excepto el nodo raíz está conectado con un arco a otro nodo k, llamado el padre de c (c es el hijo de k). El padre de un nodo, se dibuja por encima del nodo.
- Todos los nodos están conectados al nodo raíz mediante un único camino.
- Los nodos que no tienen hijos se denominan hojas, el resto de los nodos se denominan nodos interiores.

El árbol de derivación de la figura 1.4 tiene las siguientes propiedades:

- El nodo raíz está rotulado con el símbolo distinguido de la gramática.
- Cada hoja corresponde a un símbolo terminal o un símbolo no terminal.

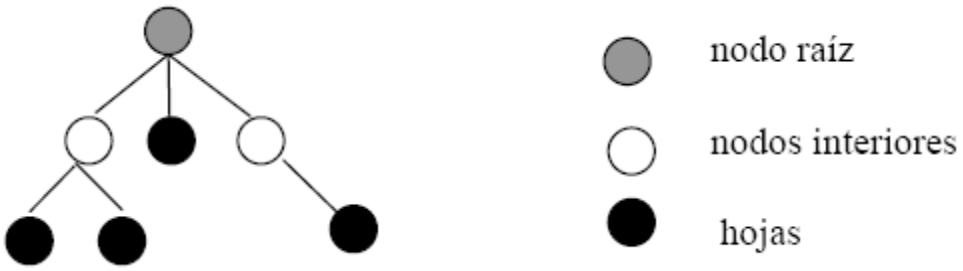


Figura 1.4: Árbol de derivación

- Cada nodo interior corresponde a un símbolo no terminal.

Las GLC tienen la propiedad de que las derivaciones pueden ser representadas en forma arborescente. Por ejemplo, considérese la gramática siguiente para producir el lenguaje de los paréntesis bien balanceados, que tienen palabras como (( )), ( ) ( ), (( ) ( )) ( ), pero no a (( ) ni) (:

1. $S \rightarrow SS$
2. $S \rightarrow (S)$
3. $S \rightarrow ( )$

Usando esta gramática, la palabra (( ) ( )) ( ) puede ser derivada de la manera que ilustra la figura 1.5. En dicha figura se puede apreciar la estructura que se encuentra implícita en la palabra (( ) ( )) ( ). A estas estructuras se les llama arboles de derivación, o también arboles de compilación – por usarse extensivamente en los compiladores – y son de vital importancia para la teoría de los compiladores de los lenguajes de programación.

1.3.1. Ambigüedad

Una gramática que produce más de un árbol de análisis sintáctico para cierto enunciado es *ambigua*. Dicho de otra forma, una gramática ambigua es aquella que produce más de una derivación por la izquierda, o más de una derivación por la derecha para el mismo enunciado.

Ejemplo: La siguiente gramática de expresiones aritméticas permite dos derivaciones por la izquierda distintas para el enunciado $id + id * id$:

1. $E \rightarrow E + T \mid E - T \mid T$
2. $T \rightarrow T * F \mid E/T \mid F$
3. $T \rightarrow (E) \mid id$

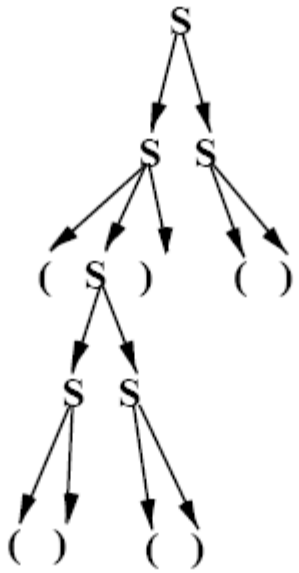


Figura 1.5: Paréntesis bien balanceados

$$
\begin{array}{rcl}
E & \Rightarrow & E + E \\
 & \Rightarrow & id + E \\
 & \Rightarrow & id + E * E \\
 & \Rightarrow & id + id * E \\
 & \Rightarrow & id + id * id
\end{array}
\qquad
\begin{array}{rcl}
E & \Rightarrow & E * E \\
 & \Rightarrow & E + E * E \\
 & \Rightarrow & id + E * E \\
 & \Rightarrow & id + id * E \\
 & \Rightarrow & id + id * id
\end{array}
$$

Los árboles de análisis sintáctico correspondientes aparecen en la figura 1.6

Observe que el árbol de análisis sintáctico de la figura 1.6(a) refleja la precedencia que se asume comúnmente para + y *, mientras que el árbol de la figura 1.6(b) no. Es decir, lo común es tratar al operador * teniendo mayor precedencia que +, en forma correspondiente al hecho de que, por lo general, evaluamos la expresión a + b * c como a + (b * c) , en vez de hacerlo como (a + b) * c.

Para la mayoría de los analizadores sintácticos, es conveniente que la gramática no tenga ambigüedades, ya que de lo contrario, no podemos determinar en forma única que árbol de análisis sintáctico seleccionar para un enunciado. En otros casos, es conveniente usar gramáticas ambiguas elegidas con cuidado, junto con reglas para eliminar la ambigüedad, las cuales "descartan" los árboles sintácticos no deseados, dejando sólo un árbol para cada enunciado.

1.3.2. Recursividad

Una gramática es recursiva si podemos hacer una derivación de un símbolo no terminal tras la cual se vuelve a tener dicho símbolo entre los símbolos de la parte derecha de la derivación.

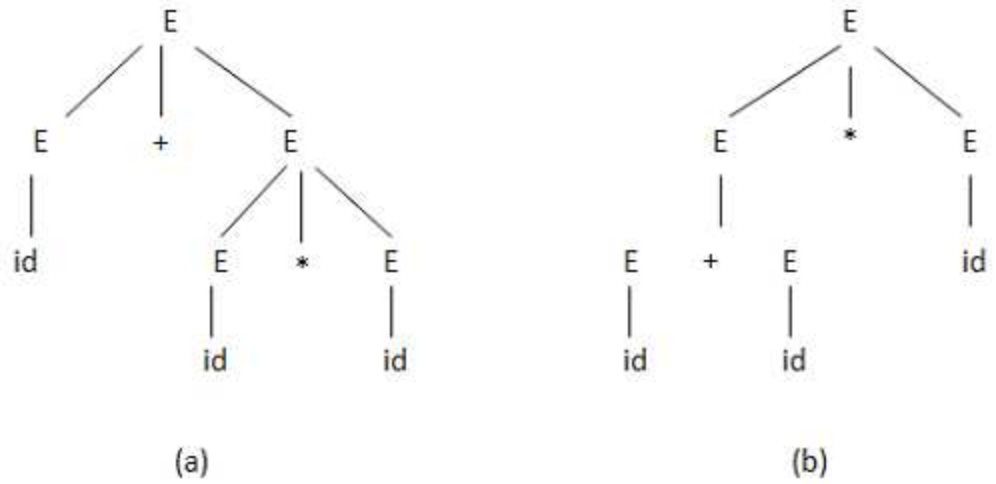


Figura 1.6: Dos árboles de análisis sintáctico para $id + id * id$

Sea:

$$A \rightarrow \alpha A \beta$$

Si se tiene:

$A \rightarrow A\beta$ se denomina recursividad por la izquierda

$A \rightarrow \alpha A$ se denomina recursividad por la derecha

1.3.3. Factorización por la izquierda

La factorización por la izquierda es una transformación gramatical, útil para producir una gramática adecuada para el análisis sintáctico predictivo, o descendente. Cuando la elección entre dos producciones A alternativas no está clara, tal vez podamos rescribir las producciones para diferir la decisión hasta haber visto la suficiente entrada como poder realizar la elección correcta.

Ejemplo: Si tenemos las siguientes producciones:

-

$$instr \rightarrow \textbf{if } expr \textbf{ then } instr \textbf{ else } instr$$

$$| \textbf{ if } expr \textbf{ then } instr$$

-

Al ver la entrada if, no podemos saber de inmediato qué producción elegir para expandir *instr*. En general, si $A \rightarrow \alpha\beta_1 \mid \alpha\beta_2$ son dos producciones A, y la entrada empieza con una cadena no vacía derivada de α, no sabemos si debemos expandir A a $\alpha\beta_1$ o a $\alpha\beta_2$. No obstante, podemos diferir la decisión si expandimos A a $\alpha A'$. Así, después de ver la entrada derivada de α, expandimos A' a β_1 o a β_2. Es decir, si se factorizan por la izquierda, las producciones originales se convierten en:

$$A \rightarrow \alpha A'$$

$$A' \rightarrow \beta_1 \mid \beta_2$$

1.4. Análisis sintáctico

El análisis sintáctico (parsing) es el proceso de determinar cómo puede generarse una cadena de terminales mediante una gramática. Al hablar sobre el problema, es más útil pensar en que se va a construir un árbol de análisis sintáctico, aun cuando un compilador tal vez no lo construya en la práctica. No obstante, un analizador sintáctico debe ser capaz de construir el árbol en principio, o de lo contrario no se puede garantizar que la traducción sea correcta.

Para cualquier gramática libre de contexto, hay un analizador sintáctico que se tarda, como máximo, un tiempo $O(n^3)$ en analizar una cadena de n terminales. Pero, por lo general, el tiempo cúbico es demasiado costoso. Por fortuna, para los lenguajes de programación reales podemos diseñar una gramática que pueda analizarse con rapidez. Los algoritmos de tiempo lineal bastan para analizar en esencia todos los lenguajes que surgen en la práctica.

Los analizadores sintácticos de los lenguajes de programación casi siempre realizan un escaneo de izquierda a derecha sobre la entrada, buscando por adelantado un terminal a la vez, y construyendo las piezas del árbol de análisis sintáctico a medida que avanzan. La mayoría de los métodos de análisis sintáctico se adaptan a una de dos clases, llamadas métodos descendente y ascendente. Estos términos se refieren al orden en el que se construyen los nodos en el árbol de análisis sintáctico.

1.4.1. Tipos de análisis sintáctico

Existen diversos algoritmos de análisis de tiempo lineal pero que no funcionan con todas las gramáticas. Estos se pueden clasificar en dos tipos: descendentes y ascendentes.

- Los **analizadores descendentes** parten del símbolo inicial de la gramática S, y van reescribiendo con las reglas de la gramática paulatinamente, a medida que se lee la entrada, construyendo así un árbol de derivación para ésta, de arriba a abajo.

- Los **analizadores ascendentes** van leyendo la entrada y reemplazando en ella algunas subpalabras que sean parte derecha de regla por la correspondiente parte

```
void A( ) {
1)      Elegir una producción A, A → X1X2...Xk;
2)      for(i = 1 a k){
3)          if ( Xi es un no terminal )
4)              llamar al procedimiento Xi();
5)          else if ( Xi es igual al símbolo de entrada actual a )
6)              avanzar la entrada hasta el siguiente símbolo;
7)          else /* ha ocurrido un error */;
        }
}
```

Figura 1.7: Procedimiento ordinario para un no terminal en un analizador sintáctico descendente

izquierda, y así sucesivamente hasta que la hemos leído entera, y nos ha quedado finalmente el símbolo inicial de la gramática S; habiendo construido así un árbol de derivación de abajo a arriba.

La clase de los algoritmos descendentes se suele llamar *LL (left-left)*, pues la entrada se lee siempre comenzando por la izquierda, y la derivación que se genera es una derivación izquierda. La clase de los algoritmos ascendentes se suele llamar *LR (left-right)*, pues la entrada se lee siempre comenzando por la izquierda, y la derivación que se genera es una derivación derecha.

1.5. Análisis descendente

El análisis sintáctico descendente puede verse como el problema de construir un árbol de análisis sintáctico para la cadena de entrada, partiendo desde la raíz y creando los nodos del árbol de análisis sintáctico en preorden (primero en profundidad). De manera equivalente, podemos considerar el análisis sintáctico descendente como la búsqueda de una derivación por la izquierda para una cadena de entrada.

En cada paso de un análisis sintáctico descendente, el problema clave es el de determinar la producción que debe aplicarse para un no terminal, por decir A. Una vez que se elige una producción A, el resto del proceso de análisis sintáctico consiste en "relación" los símbolos terminales en el cuerpo de la producción con la cadena de entrada.

1.5.1. Análisis descendente recursivo

Un programa de análisis sintáctico de descenso recursivo consiste en un conjunto de procedimientos, uno para cada no terminal. La ejecución empieza con el procedimiento para el símbolo inicial, que se detiene y anuncia que tuvo éxito si el cuerpo de su procedimiento explora toda la cadena completa de entrada. En la figura 1.7 aparece el seudocódigo para un no terminal común.

El descenso recursivo general puede requerir de un rastreo hacia atrás; es decir, tal vez requiera exploraciones repetidas sobre la entrada. Para permitir el rastreo hacia atrás, hay que modificar el código de la figura 1.7. En primer lugar, no podemos elegir una producción A única en la línea (1), por lo que debemos probar cada una de las diversas producciones en cierto orden. Después el fallo en la línea (7) no es definitivo, sino que solo sugiere que necesitamos regresar a la línea (1) y probar otra producción A.

Solo si no hay más producciones A para probar es cuando declaramos que se ha encontrado un error en la entrada. Para poder probar otra producción A, debemos restablecer el apuntador de entrada a la posición en la que se encontraba cuando llegamos por primera vez a la línea (1).

Ejemplo: Considere la siguiente gramática:

$$S \rightarrow c\,A\,d$$

$$A \rightarrow a\,b \mid a$$

Para construir un árbol de análisis sintáctico descendente para la cadena de entrada $w = cad$, empezamos con un árbol que consiste en un solo nodo etiquetado como S, y el apuntador de entrada apunta a c, el primer símbolo de w. S sólo tiene una producción, por lo que la utilizamos para expandir S y obtener el árbol de la figura 1.8(a). La hoja de la izquierda, etiquetada como c, coincide con el primer símbolo de la entrada w, por lo que avanzamos el apuntador de entrada hasta a, el segundo símbolo de w, y consideramos la siguiente hoja, etiquetada como A.

Ahora expandimos A mediante la primera alternativa $A \rightarrow a\,b$ para obtener el árbol de la figura 1.8(b). Tenemos una coincidencia para el segundo símbolo de entrada a, por lo que avanzamos el apuntador de entrada hasta d, el tercer símbolo de entrad, y comparamos a d con la siguiente hoja etiquetada con b. Como b no coincide con d, reportamos un error y regresamos a A para ver si hay otra alternativa para A que no hayamos probado y que pueda producir una coincidencia.

Al regresar a A, debemos restablecer el apuntador de entrada a la posición 2, la posición que tenía cuando llegamos por primera vez a A, lo cual significa que el procedimiento para A debe almacenar el apuntador de entrada en una variable local.

La segunda alternativa para A produce el árbol de la figura 1.8(c). La hoja a coincide con el segundo símbolo de w y la hoja d coincide con el tercer símbolo. Como hemos producido un árbol de análisis sintáctico para w, nos detenemos y anunciamos que se completó el análisis sintáctico con éxito.

1.5.2. Análisis descendente predictivo

El objetivo del análisis predictivo es la construcción de un analizador sintáctico descendente que nunca retrocede; y para ello este intenta encontrar entre las producciones de la gramática una derivación por la izquierda del símbolo inicial para una cadena de

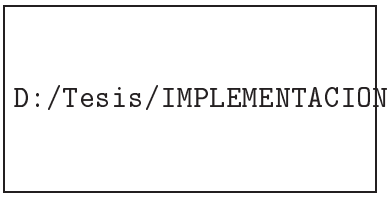


Figura 1.8: Los pasos de un análisis sintáctico descendete

entrada. Para que el algoritmo tenga una complejidad lineal, el analizador debe realizar una predicción de la regla a aplicar.

Esto implica, se debe conocer, dado el token de la entrada, a, que esté siendo analizado, y el no terminal a expandir A, cuál de las alternativas de producción $A \rightarrow a_1 \,|\, a_2 \,|\, ... \, a_n$ es la única posible que da lugar a que el resto de la cadena que se está analizando empiece por a. Dicho de otra forma, la alternativa apropiada debe poderse predecir sólo con ver el primer símbolo que produce.

Este tipo de analizadores recurren al principio de de previsión, el cual consiste en "observar" un carácter de la palabra de entrada que aún no ha sido leído (esto se llama en ingles "lookahead", mirar hacia adelante). El carácter leído por adelantado nos permite en algunas ocasiones elegir adecuadamente cual de las reglas de producción utilizar basándose fuertemente en gramáticas LL(1) las cuales hacen posible el no retroceso de estos analizadores predictivos , lo que los hace mas eficientes en cuestiones de velocidad de procesamiento.

1.6. Gramáticas LL(1)

Los analizadores sintácticos predictivos, es decir, los analizadores sintácticos de descenso recursivo que no necesitan rastreó hacia atrás, pueden construirse para una clase de gramáticas llamadas LL(1). La primera "L" en LL(1) es para explorar la entrada de izquierda a derecha (por left en inglés), la segunda "L" para producir una derivación por la izquierda, y el "1" para usar un símbolo de entrada de anticipación en cada paso, para tomar las decisiones de acción del análisis sintáctico.

La clase de gramáticas LL(1) es lo bastante robusta como para cubrir la mayoría de las construcciones de programación, aunque hay que tener cuidado al escribir una gramática adecuada para el lenguaje fuente. Por ejemplo, ninguna gramática recursiva por la izquierda o ambigua puede ser LL(1).

1.6.1. Conjuntos PRIMEROS y SIGUIENTES

La construcción de los analizadores sintácticos descendentes y ascendentes es auxiliada por dos funciones, PRIMEROS y SIGUIENTES nos permiten elegir la producción que vamos a aplicar, con base en el siguiente símbolo de entrada.

1.7. Java

Java es toda una tecnología orientada al desarrollo de software con el cual podemos realizar cualquier tipo de programa. Hoy en día, la tecnología Java ha cobrado mucha importancia en el ámbito de Internet gracias a su plataforma J2EE. Pero Java no se queda ahí, ya que en la industria para dispositivos móviles también hay una gran acogida para este lenguaje.

La tecnología Java está compuesta básicamente por 2 elementos: el lenguaje Java y su plataforma. Con plataforma nos referimos a la máquina virtual de Java (Java Virtual Machine).

Una de las principales características que favoreció el crecimiento y difusión del lenguaje Java es su capacidad de que el código funcione sobre cualquier plataforma de software y hardware. Esto significa que nuestro mismo programa escrito para Linux puede ser ejecutado en Windows sin ningún problema. Además es un lenguaje orientado a objetos que resuelve los problemas en la complejidad de los sistemas, entre otras.

Capítulo 2

Análisis del sistema

Los algoritmos son la clave para el buen entendimiento y desarrollo de éste sistema, ahora, sin mas preámbulos nos adentramos en el análisis de algoritmos necesarios.

2.1. Análisis de Requerimientos

Existe la necesidad de analizar de manera acertada y ejemplificada cada uno de los algoritmos que serán utilizados para el desarrollo del sistema. Sera necesario un pequeño editor con funciones básicas (nuevo, abrir y guarda), esto p/ hacer mas cómodo para el usuario el proceso de análisis de la gramática.

Así también es necesario el desarrollo de una gramática la cual pueda analizar las gramáticas; debe ser definida la sintaxis la cual soportara el analizador.

1. Después del proceso de edición de gramática, el sistema debe aplicar los algoritmos siguientes:
2. Algoritmo de eliminación de recursividad por la izquierda
3. Algoritmo de factorización por la izquierda
4. Calculo de PRIMEROS y SIGUIENTES
5. Calculo de PREDICTIVO y verificación de la condición LL1

2.2. Alcances

Desarrollar un sistema que edite gramáticas y permita verificar si cumplen las características necesarias de una gramática LL1, cabe mencionar que si esta condición se cumple, dicha gramática puede ser implementada como analizador sintáctico descendente predictivo.

Se pretende que el sistema obtenido genere más interés en el desarrollo de compiladores, ya que la aplicación de estos es multidisciplinaria. También hacer énfasis en que la sintaxis determina como se debe ver un programa y se relaciona mutuamente

con su representación textual o estructura de tal forma que tenga una definición precisa, por tales motivos el análisis desde el punto de vista sintáctico se prevé como una necesidad para los compiladores.

Esta herramienta a desarrollar deberá tener la funcionalidad requerida para ser fácil de usar capaz de obenter resultados claros y concisos para alcanzar la meta de aprendizaje-enseñanza.

2.3. Métodos propuestos

A continuación se enlista el orden de los procesos que se consideran necesarios para lograr llevar a cabo esta investigación y obtención de resultados.

- Aprender, entender y analizar elementos teóricos necesarios en la aplicación del desarrollo y análisis de gramáticas.
- Investigar, identificar y poner en claro los puntos clave para el desarrollo de los analizadores sintácticos.
- Analizar los algoritmos de eliminación y factorización por la izquierda para después obtener los conjuntos de primeros, siguientes y predictivos, programarlos y realizar pruebas.
- Modelar las posibles interfaces que se pueden aplicar al sistema.
- Ajustar los algoritmos programados junto con las interfaces y realizar pruebas.
- Observar los resultados obtenidos así como las conclusiones.

2.4. Limitaciones

- El sistema no pretende corregir automáticamente las gramáticas introducidas, sino más bien es una herramienta la cual se encargara de brindar apoyo con la implementación de los algoritmos necesarios para la corrección de éstas gramáticas.
- Todo sistema que trabaja con gramáticas tiende a tener errores de ambigüedad que en ocasiones suelen ser difíciles de hallar, llamados comúnmente errores lógicos, este sistema no será la excepción y está expuesto a ellos.
- El sistema únicamente verificará gramáticas para analizadores descendentes sintácticos predictivos; conocidas comúnmente como gramáticas LL1.

2.5. Escritura de una gramática

Las gramáticas son capaces de describir casi la mayoría de la sintaxis de los lenguajes de programación. Por ejemplo, el requerimiento de que los identificadores deben declararse antes de usarse, no puede describirse mediante una gramática libre de contexto. Por lo tanto las secuencias de los tokens que acepta un analizador sintáctico forman un superconjunto del lenguaje de programación; las fases siguientes del compilador deben analizar la salida del analizador sintáctico, para asegurar que cumpla las reglas que no verifica el analizador sintáctico. Consideramos varias transformaciones que podrían aplicarse para obtener una gramática más adecuada para el análisis sintáctico. Una técnica puede eliminar la ambigüedad en la gramática, y las otras (eliminación de recursividad por la izquierda y factorización por la izquierda) son útiles para rescribir las gramáticas, de manera que sean adecuadas para el análisis sintáctico descendente.

2.6. Eliminación de la recursividad por la izquierda

Una gramática es recursiva por la izquierda si tiene un no-terminal A tal que existe una producción $A \Rightarrow^+ A\alpha$. Los métodos de análisis sintáctico descendentes no pueden manejar las gramáticas recursivas por las izquierda, por lo que se necesita una transformación para eliminar la recursividad por la izquierda.

Algoritmo para eliminar recursividad a la izquierda:

- **Primer paso:** se agrupan todas las producciones de A en la forma:

$$A \to A\alpha_1 \mid A\alpha_2 \mid ... \mid A\alpha_m \mid \beta_1 \mid \beta_2 \mid ... \mid \beta_n$$

  donde ninguna β_i comienza con una A.

- **Segundo paso:** se sustituyen las producciones de A por:

$$A \to \beta_1 A' \mid \beta_2 A' \mid ... \mid \beta_n A'$$

$$A' \to \alpha_1 A' \mid \alpha_2 A' \mid ... \mid \alpha_m \mid \varepsilon$$

Nota: Este método elimina en forma sistemática la recursividad por la izquierda de una gramática. Se garantiza que funciona si la gramática no tiene ciclos (derivaciones de la forma $A \Rightarrow^+ A$) o producciones $A \to \varepsilon$).

Ejemplo: Eliminar la recursividad a la izquierda de:

1. $E \rightarrow E + T \mid T$
2. $T \rightarrow T * F \mid F$
3. $E \rightarrow (E) \mid num \mid id$

En la producción uno se hace notar que no existen ciclos ni producciones vacías , ahora se aplica el primer paso:

$$E \rightarrow E\alpha \mid \beta \text{ donde } \alpha = +T \text{ y } \beta = T$$

Luego aplicamos segundo paso:

$$E \rightarrow \beta E' \text{ y } E' \rightarrow \alpha E' \mid \varepsilon$$

Obtenemos:

$$E \rightarrow TE' \text{ y } E' \rightarrow +TE' \mid \varepsilon$$

Así también aplicando el mismo procedimiento a la producción número dos, obtenemos la siguiente gramática final no recursiva:

1. $E \rightarrow TE'$
2. $E' \rightarrow +TE' \mid \varepsilon$
3. $T \rightarrow FT'$
4. $T' \rightarrow *FT' \mid \varepsilon$
5. $F \rightarrow (E) \mid id$

2.7. Factorización por la izquierda

La factorización por la izquierda es una transformación gramatical, útil para producir una gramática adecuada para el análisis sintáctico predictivo, o descendente. Cuando la elección entre dos producciones A alternativas no está clara, tal vez podamos rescribir las producciones para diferir la decisión hasta haber visto la suficiente entrada como para poder realizar la elección correcta. Por ejemplo:

$$\begin{array}{lcl} Stmt & \rightarrow & \text{if } E \text{ then } Stmt \text{ else } Stmt \\ & & \mid \text{if } E \text{ then } Stmt \\ & & \mid \text{Otras} \end{array}$$

Si en la entrada tenemos el token if no se sabe cual de las dos alternativas elegir para expandir . La idea es reescribir la producción para retrasar la decisión hasta haber visto de la entrada lo suficiente para poder elegir la opción correcta.

$$\begin{array}{lcl} Stmt & \rightarrow & \text{if } E \text{ then } Stmt\ Stmt' \mid \text{Otras} \\ Stmt\text{'} & \rightarrow & \text{else } Stmt \mid \varepsilon \end{array}$$

Algoritmo para factorizar la gramática a la izquierda:

- **Primer paso:** para cada no-terminal buscar el prefijo más largo común a dos o más alternativas de A. $A \rightarrow \alpha\beta_1 \mid \alpha\beta_2 \mid ... \mid \alpha\beta_n \mid \gamma$
- **Segundo paso:** Si $\alpha \neq \varepsilon$, sustituir todas las producciones de A, de la forma $A \rightarrow \alpha\beta_1 \mid \alpha\beta_2 \mid ... \mid \alpha\beta_n \mid \gamma$, donde γ representa todas las alternativas de A que no comienzan con α por:

$$A \rightarrow \alpha A' \mid \gamma$$

$$A' \rightarrow \beta_1 \mid \beta_2 \mid ... \mid \beta_n$$

Ejemplo: Factorización a la izquierda de:

1. $T \rightarrow PmR \mid PmD$
2. $P \rightarrow amb \mid amd$
3. $D \rightarrow d$
4. $R \rightarrow r$

Se aplica el primer paso a la producción uno:

$$T \rightarrow \alpha\beta_1 \mid \alpha\beta_2 \text{ donde } \alpha = Pm,\ \beta_1 = R,\ \beta_2 = D$$

Luego aplicamos segundo paso:

$$T \rightarrow \alpha T' \text{ y } T' \rightarrow \beta_1 \mid \beta_2$$

Obtenemos:

$$T \rightarrow PmT\text{'} \text{ y } T' \rightarrow R \mid D$$

Así también aplicando el mismo procedimiento a la producción número dos, obtenemos la siguiente gramática final con factorización :

1. $T \rightarrow PMT'$
2. $T' \rightarrow R \mid D$
3. $P \rightarrow amP'$
4. $P' \rightarrow b \mid d$
5. $D \rightarrow d$
6. $R \rightarrow r$

2.8. Conjuntos de predicción

Son conjuntos de tokens que ayudan a predecir qué regla se debe aplicar para la variable que hay que derivar. Se construyen a partir de los símbolos de las partes derechas de las producciones de la gramática. El analizador consulta el siguiente token en la entrada y si pertenece al conjunto de predicción de una regla (de la variable que hay que derivar), aplica esa regla. Si no puede aplicar ninguna regla, se produce un de error.

2.8.1. Cálculo de conjuntos de predicción

Los conjuntos de predicción de una regla se calculan en función de los primeros símbolos que puede generar la parte derecha de esa regla, y a veces (cuando esa parte genera la cadena vacía) en función de los símbolos que pueden aparecer a continuación de la parte izquierda de la regla en una forma sentencial.

2.8.2. *PRIMEROS*

Si α es una forma sentencial compuesta por una concatenación de símbolos, $PRIMEROS(\alpha)$ es el conjunto de terminales (o ε) con los que puede empezar la cadena derivable (en cero o más pasos) a partir de su argumento.

Por ejemplo, para la GLC con reglas $S \to aA$, $A \to Sb$, $A \to b$, nos damos cuenta de que las cadenas que se pueden derivar a partir de S tienen que empezar con a, por que lo único que puede producir S es aA, que empieza con a. Por ello $PRIMEROS(S) = \{a\}$.

El conjunto $PRIMEROS$ puede referirse a una forma sentencial ($PRIMEROS(\alpha)$, siendo α por ejemplo aBD), pero también a un símbolo No Terminal de la gramática considerada ($PRIMEROS(A)$). Si la gramática está formada por reglas $A \to \alpha_i$ el conjunto $PRIMEROS(A)$ será la unión de todos los $PRIMEROS(\alpha_i)$, que deben ser calculados por separado.

Para calcular $PRIMEROS(\alpha)$ basta con seguir las siguientes reglas para cada regla de producción de la gramática:

1. Si $\alpha \equiv \varepsilon$, donde ε es la cadena vacía :

   Añadir $\{\varepsilon\}$ a $PRIMEROS(\alpha)$

2. Si $\alpha \equiv a$, donde a es un Terminal:

   Añadir $\{a\}$ a $PRIMEROS(\alpha)$

3. Si $\alpha \equiv A$, donde A es un No Terminal:

   Añadir $PRIMEROS(A)$ a $PRIMEROS(\alpha)$.

4. Si $\alpha \equiv AB$, donde A y B son No Terminales y $PRIMEROS(A)$ incluye $\{\varepsilon\}$:

   Añadir $PRIMEROS(A)$ a $PRIMEROS(\alpha)$, pero sin incluir de momento $\{\varepsilon\}$.

   Además, y puesto que A puede ser vacío, añadir $PRIMEROS(B)$ a $PRIMEROS(\alpha)$.

   Esta regla puede extenderse a todos los No Terminales que aparezcan detrás de A. Por ejemplo, si $\alpha \equiv ABCD$ y $PRIMEROS(A)$, $PRIMEROS(B)$ y $PRIMEROS(C)$ incluyen $\{\varepsilon\}$, entonces habrá que añadir a $PRIMEROS(\alpha)$ la unión de $PRIMEROS(A)$, $PRIMEROS(B)$, $PRIMEROS(C)$ y $PRIMEROS(D)$, sin incluir $\{\varepsilon\}$.

   Por último se decide si incluir $\{\varepsilon\}$ en $PRIMEROS(\alpha)$: sólo debe incluirse si aparece en los conjuntos $PRIMEROS$ de todos los No Terminales. Por ejemplo en el caso anterior, si $PRIMEROS(D)$ también incluyese $\{\varepsilon\}$, entonces $PRIMEROS(\alpha)$ incluiría $\{\varepsilon\}$.

2.8.3. Ejemplo de cálculo de *PRIMEROS*

Veamos un ejemplo de cálculo detallado de $PRIMEROS$ para la siguiente gramática, que describe expresiones aritméticas con sumas y multiplicaciones:

1. $E \to TE'$

2. $E' \rightarrow +TE' \mid \varepsilon$
3. $T \rightarrow FT'$
4. $T' \rightarrow *FT' \mid \varepsilon$
5. $F \rightarrow (E) \mid ident$

- $PRIMEROS(E) = PRIMEROS(TE')$ (en adelante escribiremos simplemente $PRIMEROS(E)$).

  $PRIMEROS(E) = PRIMEROS(T)$ por la Regla 3, así que pasamos a calcular $PRIMEROS(T)$.
- $PRIMEROS(T) = PRIMEROS(F)$ por la Regla 3, así que pasamos a calcular $PRIMEROS(F)$.
- $PRIMEROS(F) = \{$ (, $ident\}$ por la Regla 2.
- En este punto ya sabemos que $PRIMEROS(E)$ =$PRIMEROS(T)$= $PRIMEROS(F)$ =$\{$ (, $ident\}$
- $PRIMEROS(E') = \{+, \varepsilon\}$ por las Reglas 2 y 1.
- $PRIMEROS(T') = \{*, \varepsilon\}$ por las Reglas 2 y 1.

  -

Resultado:

-

$$PRIMEROS(E) = \{ (\ , ident\}$$

$$PRIMEROS(E') = \{+, \varepsilon\}$$

$$PRIMEROS(T) = \{ (\ , ident\}$$

$$PRIMEROS(T') = \{*, \varepsilon\}$$

$$PRIMEROS(F) = \{ (\ , ident\}$$

2.8.4. *SIGUIENTES*

Si A es un símbolo No Terminal de una gramática, $SIGUIENTES(A)$ es el conjunto de Terminales (incluyendo el símbolo de fin de cadena, \$) que pueden aparecer justo después de A en alguna forma sentencial derivada del símbolo inicial.

El cálculo de conjuntos $SIGUIENTES$ se realiza a partir de estas reglas:

1. Los conjuntos $SIGUIENTES$ de todos los No Terminales son inicialmente vacíos, excepto el del No Terminal inicial de la gramática, en el que se incluye el símbolo $\{\$\}$.

   A partir de este punto, para calcular cada uno de los conjuntos deben aplicarse las dos siguientes reglas (no son excluyentes) por cada ocurrencia del No Terminal en la parte derecha de alguna regla de producción:

2. Si $A \rightarrow \alpha B \beta$:

   Añadir a $SIGUIENTES(B)$ los elementos de $PRIMEROS(\beta)$, con la excepción de $\{\, \varepsilon \,\}$: este símbolo nunca se incluirá en los conjuntos $SIGUIENTES$.

3. Si $A \rightarrow \alpha B$, o bien $A \rightarrow \alpha B \beta$ donde $PRIMEROS(\beta)$ contiene $\{\, \varepsilon \,\}$:

   Añadir a $SIGUIENTES(B)$ los elementos de $SIGUIENTES(A)$.

Conviene comprender las reglas en lugar de intentar memorizarlas: en el caso de $SIGUIENTES$, se comprueba intuitivamente que los posibles Terminales que pueden aparecer siguiendo a un No Terminal B son:

- Los que pertenecen al conjunto $PRIMEROS$ del No Terminal que aparece inmediatamente después que el No Terminal B. Esto lo expresa la Regla 2.
- Si el No Terminal B aparece el último en la parte derecha de una regla de producción (o bien el No Terminal que hay a su derecha puede derivar la cadena vacía), estaremos en un caso como el siguiente:

  1. $S \rightarrow aAb$
  2. $A \rightarrow cB$
  3. $B \rightarrow ...$

  Intuitivamente, ¿cuál es el conjunto de Terminales que pueden seguir a B en una cadena? Como B aparece el último en la parte derecha de una regla de producción de A, habrá que buscar los Terminales que puedan seguir a A. Es decir, los siguientes de B serán, como mínimo, los siguientes de A, en este caso $\{b\}$. Esto es lo que expresa la Regla 3.

2.8.5. Ejemplo de cálculo de $SIGUIENTES$

Veamos ejemplos de cálculo detallado de $SIGUIENTES$ para las mismas gramáticas que utilizamos antes:

1. $E \rightarrow TE'$
2. $E' \rightarrow +TE' \mid \varepsilon$
3. $T \rightarrow FT'$
4. $T' \rightarrow *FT' \mid \varepsilon$
5. $F \rightarrow (E) \mid ident$

Partimos de los conjuntos $PRIMEROS$ calculados previamente:
$PRIMEROS(E) = \{ (, ident\}$
$PRIMEROS(E') = \{+, \varepsilon\}$
$PRIMEROS(T) = \{ (, ident\}$
$PRIMEROS(T') = \{*, \varepsilon\}$
$PRIMEROS(F) = \{ (, ident\}$

- $SIGUIENTES(E) = \{\$\}$ por la Regla 1.
- $SIGUIENTES(E) = SIGUIENTES(E) \cup \{) \} = \{\$,) \}$ por la Regla 2: hemos buscado E en las partes derechas de las reglas y hemos encontrado $F \rightarrow (E)$, comprobando que $PRIMEROS( )) = \{) \}$. Como E no aparece en la parte derecha de ninguna otra regla, damos por cerrado su conjunto $SIGUIENTES$.
- $SIGUIENTES(E') = SIGUIENTES(E) = \{\$,) \}$ por la Regla 3: E' aparece el último en la parte derecha de dos reglas de producción, aunque el No Terminal que las origina es el mismo: E. Por tanto, se añade $SIGUIENTES(E)$ a $SIGUIENTES(E')$ y, como E' no aparece más, se da por cerrado su conjunto.
- $SIGUIENTES(T) = PRIMEROS(E') \cup SIGUIENTES(E') = \{+, \$,) \}$ por las Reglas 2 y 3. Hemos aplicado la Regla 3 porque en $E \rightarrow +TE'$ el No Terminal T podría ser el último a la derecha, ya que E' deriva la palabra vacía.
- $SIGUIENTES(T') = SIGUIENTES(T) = \{+, \$,) \}$ por la Regla 3.
- $SIGUIENTES(F) = PRIMEROS(T') \cup SIGUIENTES(T) \cup SIGUIENTES(T') = \{*, +, \$,) \}$ por las Reglas 2 y 3.

-

Resultado:

$$SIGUIENTES(E) = \{\$,) \}$$

$$SIGUIENTES(E') = \{\$,) \}$$

$$SIGUIENTES(T) = \{+, \$,)\}$$

$$SIGUIENTES(T') = \{+, \$,)\}$$

$$SIGUIENTES(F) = \{*, +, \$,)\}$$

2.8.6. *PREDICT*

La función *PREDICT* se aplica a producciones de la gramática $A \rightarrow a$ y devuelve un conjunto, llamado *conjunto de predicción*, que puede contener cualesquiera terminales de la gramática y el símbolo *"$"*, pero nunca puede contener ε.

Reglas para el cálculo del conjunto *PREDICT*:

$PREDICT(A \rightarrow \alpha) = Si\ \varepsilon\ {}_{\epsilon}PRIMEROS(\alpha)\ entonces\ = (PRIMEROS(\alpha) - \{\varepsilon\}) \cup$
$(SIGUIENTES(A))\ sino = PRIMEROS(\alpha)$

Por ejemplo:
Supóngase la siguiente gramática:

$S \rightarrow AB \mid s$
$A \rightarrow aSc \mid eBf \mid \varepsilon$
$B \rightarrow bAd \mid \varepsilon$

Calculamos los conjuntos de predicción utilizando la regla adecuada en cada caso:

$PREDICT(S \rightarrow AB) = (PRIMEROS(AB) - \{\varepsilon\}) \cup SIGUIENTES(S)$
$= \{a, e, b, c, \$\}$
$PREDICT(S \rightarrow s) = PRIMEROS(s) = \{s\}$
$PREDICT(A \rightarrow aSc) = PRIMEROS(aSc) = \{a\}$
$PREDICT(A \rightarrow eBf) = PRIMEROS(eBf) = \{e\}$
$PREDICT(A \rightarrow \varepsilon) = (PRIMEROS(\varepsilon) - \{\varepsilon\}) \cup SIGUIENTES(A) = \{b, d, c, \$\}$
$PREDICT(B \rightarrow bAd) = PRIMEROS(bAd) = \{b\}$
$PREDICT(B \rightarrow \varepsilon) = (PRIMEROS(\varepsilon) - \{\varepsilon\}) \cup SIGUIENTES(B) = \{f, c, \$\}$

2.9. Gramáticas LL(1)

Los analizadores sintácticos predictivos, es decir los analizadores sintácticos de descenso recursivo que no necesitan rastreo hacia atrás, pueden construirse para una clase de gramáticas llamadas LL(1). La primera "L" en LL(1) es para explorar la entrada de izquierda a derecha (por left en inglés), la segunda "L" para producir una derivación por la izquierda, y el "1" para usar un símbolo de entrada de anticipación en cada paso, para tomar las decisiones de acción del análisis sintáctico. Pueden

construirse analizadores sintácticos predictivos para las gramáticas LL(1), ya que puede seleccionarse la producción apropiada a aplicar para una no terminal con sólo analizar el símbolo de entrada actual.

Características que se deben cumplir:

- Una gramática LL(1) no puede ser recursiva a la izquierda y debe estar factorizada.
- Una gramática LL(1) es no ambigua.

2.9.1. La condición LL(1)

Para que una gramática pertenezca al conjunto de gramáticas LL(1) ha de cumplir la condición LL(1).

Para que la regla a aplicar sea siempre única, se debe exigir que los conjuntos de predicción de las reglas de cada no terminal sean disjuntos entres sí.

Si la gramática es LL(1) y se puede realizar su análisis sintáctico en tiempo lineal.

La condición LL(1) es necesaria y suficiente para poder construir un analizador sintáctico predictivo para una gramática.

Condición LL(1):

Dadas todas las producciones de la gramática para un mismo terminal:
$A \rightarrow \alpha_1|\alpha_2|...|\alpha_n \; \forall A \epsilon N$
se debe cumplir la siguiente condición:
$\forall i, j \; (i \neq j) \; PREDICT(A \rightarrow \alpha_i) \; \cap PREDICT(A \rightarrow \alpha_j) = \oslash$

Capítulo 3

Diseño del sistema

Una vez mostrados los algoritmos necesarios en el capítulo anterior es hora de entrar en términos de diseño para después mapearlos a la implementación, en este capítulo se muestra como será llevado a cabo el análisis de las gramáticas y la muestra del prototipo para el sistema.

3.1. Diseño de gramáticas

Como se identifico en el marco teórico es necesaria la revisión tanto léxica como sintáctica, al revisar de forma léxica se va agrupar en tokens, estos obtenidos de la gramática de entrada, para que en el siguiente paso se desglose y corrobore que en realidad están ubicados en orden correspondiente.

3.1.1. Proceso de Análisis

Una vez que se han definido las gramáticas a utilizar en el ámbito de reconocimiento, se muestra en la figura 3.1 el procedimiento que será llevado a cabo para el buen análisis y búsqueda de errores al mismo tiempo que se leen los caracteres.

En ésta figura se aprecia como se van tomando los caracteres del flujo de entrada, el análisis léxico empacará los tokens según lo especifica la gramática, si esto no ocurre, inmediatamente se disparan mensajes de error. Obviamente los espacios en blanco, tabuladores así como saltos de línea serán ignorados conforme se avance.

Ahora, el análisis sintáctico será quién solicite los tokens a ser analizados, una vez que se ha empacado un token sin error, llega al proceso sintáctico, sí el tipo de token que se requería se es obtenido, se vuelve a iniciar el proceso de solicitud y así sucesivamente hasta culminar la gramática de entrada.

3.2. Análisis Léxico

Se conoce también como análisis lineal debido a que se trata de un barrido secuencial del código fuente. La función de un analizador léxico es la de identificar a los elementos

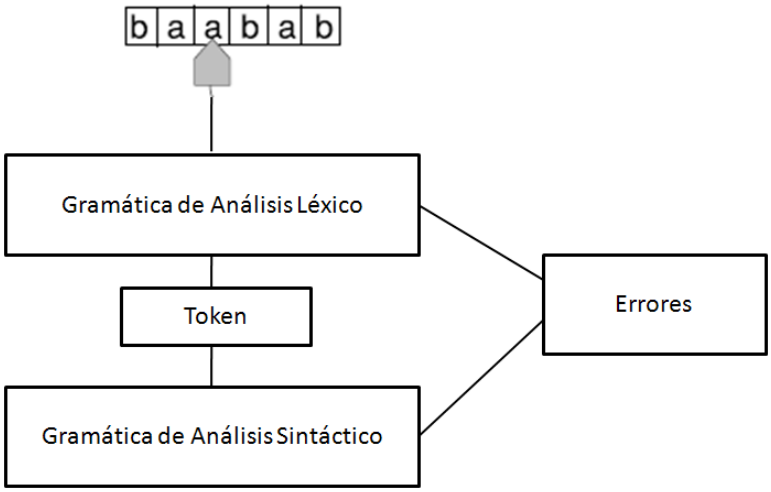


Figura 3.1: Proceso de Análisis de Gramática

sintácticos básicos del lenguaje los cuales son indivisibles y a los que llamamos tokens. El analizador debe de ignorar caracteres blancos (espacios, tabuladores y retornos de carro) y reconocer a la cadena mas larga de caracteres que forme un token valido. Por ejemplo, debe decidir que la cadena "for3" es un token del tipo identificador y no dos identificadores del tipo palabra reservada (for) seguida de otro token del tipo constante entera (3). El analizador léxico entrega el tipo de token de que se trata mediante una constante predeterminada y la cadena leída a la cual por cierto llamamos lexema. Una manera preferida de entregar el lexema es insertándolo en una tabla conocida como "Tabla de símbolos" y entregando la posición que el lexema ocupa en esa tabla. El analizador éxico es llamado por el analizador sintáctico cada vez que este requiere de otro token.

3.2.1. Gramática de Análisis Léxico

Con esta gramática se desea lograr un análisis léxico revisando carácter por carácter de entada y agrupándolos de tal forma que se armen tokens para que sean devueltos hacía el analizador sintáctico.

1. $Term \rightarrow MinPalabra \mid "Palabra" \mid < Palabra > \mid Num \mid Sim$
2. $NTerm \rightarrow MayPalabra$
3. $Palabra \rightarrow MayPalabra \mid MinPalabra \mid NumPalabra \mid \varepsilon$
4. $Num \rightarrow 0 \mid 1 \mid ... \mid 8 \mid 9$
5. $May \rightarrow A \mid B \mid C \mid ... \mid X \mid Y \mid Z$
6. $Min \rightarrow a \mid b \mid c \mid ... \mid x \mid y \mid z$

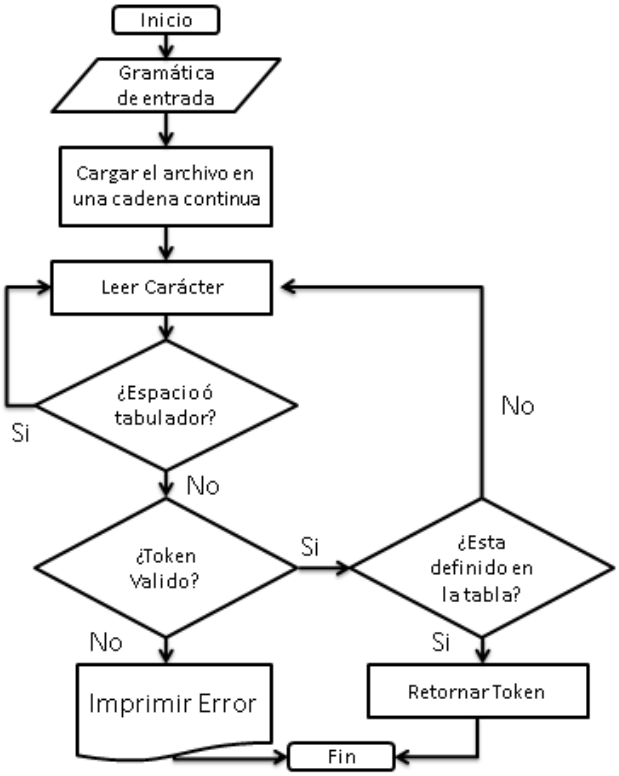


Figura 3.2: Diagrama de análisis léxico.

7. $Sim \rightarrow + \mid - \mid \$ \mid ... \mid / \mid * (sin\ ;)$

3.2.2. Diagrama de Análisis Léxico

En lo que corresponde al analizador léxico, la rutina a seguir para este proceso se muestra en la figura 3.2. Como entrada se necesita la gramática en forma de archivo, para ser procesado correctamente como si se tratase de una cinta lectora, el archivo de entrada se asignara a una cadena lineal para avanzar y retroceder sin complicación alguna, eliminando así los saltos de línea. Deben existir funciones de avance y retroceso para facilitar el desplazamiento del analizador.

Una vez establecida la gramática en una cadena lineal procede a la lectura de carácter por carácter, si en el camino se encuentra algún espacio en blanco o tabulador se ignora y procede a leer el siguiente carácter, esto se hará de forma continua si es que existen espacios contiguos. Cada carácter leído será adjuntado al token y una vez verificada su concordancia con los símbolos ya definidos, en una tabla de símbolos definida previamente, se procede a evaluar que el token sea igual a alguno definido en la tabla o por lo menos sea parte del contexto de algún token. Si es igual a algún token de la tabla, será retornado, sino, se procede a leer otro carácter.

Si el token no pertenece en ningún sentido a los definidos en la tabla, obviamente se marcara un error y el proceso de análisis se detendrá.

3.3. Análisis Sintáctico

Un analizador sintáctico es una de las partes de un compilador que transforma su entrada en un árbol de derivación. Un analizador léxico crea tokens de una secuencia de caracteres de entrada y son estos tokens los que son procesados por el analizador sintáctico para construir la estructura de datos, por ejemplo un árbol de análisis o árboles de sintaxis abstracta.

3.3.1. Gramática de Análisis Sintáctico

La gramática de análisis sintáctico se orienta a descubrir errores de orden ó tipo de token que se esperaba, estos errores sobresaldrán cuando la gramática especifique determinado token y éste no aparezca al ser solicitado.

1. $Gramatica \rightarrow G\{Bloque$
2. $Bloque \rightarrow ConsVarIniReglas\}$
3. $Cons \rightarrow constantes\{TermT$
4. $Var \rightarrow variables\{NTermNT$
5. $Ini \rightarrow inicial\{NTerm\}$
6. $Reglas \rightarrow reglas\{Prod$
7. $T \rightarrow, TermT \mid \}$
8. $NT \rightarrow, NTermNT \mid \}$
9. $TP \rightarrow\} \mid Prod$
10. $Prod \rightarrow NTerm - - > CT$
11. $Cuerpo \rightarrow TCuerpo \mid NTCuerpo \mid \mid CT \mid ; TP$
12. $CT \rightarrow TCuerpo \mid NTCuerpo \mid \epsilon Vacio$
13. $Vacio \rightarrow \mid CT \mid ; TP$

3.3.2. Diagrama de Análisis Sintactico

Se ha descrito en secciones anteriores el funcionamiento del analizador léxico y justamente se hace mención de la cooperación que hace al analizador sintáctico. En el diagrama de flujo de la figura 3.3, observamos que se le proporciona la gramática correspondiente al analizador sintáctico con el fin de que éste se encargue de darle una copia al analizador léxico y se encargue de seccionarlo en tokens para poder ser procesados.

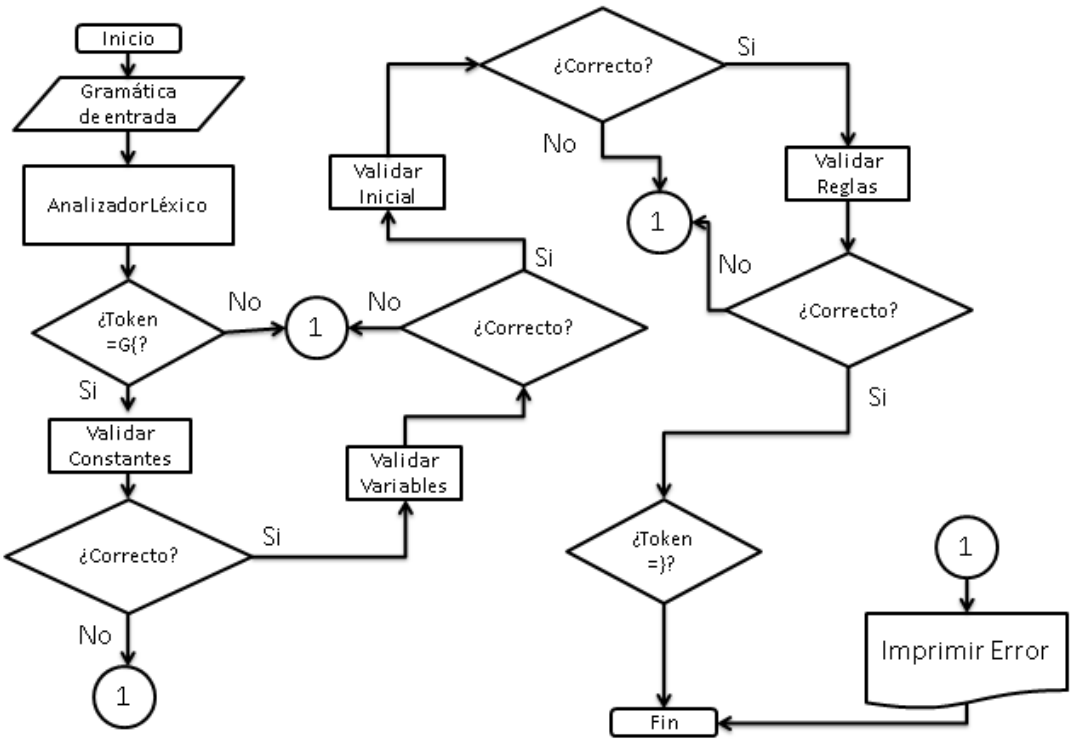


Figura 3.3: Diagrama de análisis sintáctico.

Se le dará la orden al analizador léxico para que retorne un token, si la secuencia inicial de tokens es "G{" (que significa apertura de gramática) se continua con el procedimiento, en caso contrario se lanzara un error. Ahora se evalúan las constantes, ¿Qué se evalúa?- Esto será descrito más adelante por ahora solo se tomaran como cajas de proceso donde se les introduce un dato y devuelven un resultado. Si la evaluación de las constantes es incorrecta se lanza un error, de lo contrario se continua con la validación de variables, un proceso similar al de las constantes, al retornar su veredicto revisamos si fue incorrecto entonces lanzamos un error; si fue correcto entramos a la siguiente función de análisis sobre las reglas de producción. Si el análisis fue incorrecto se imprime el error, de lo contrario continua a pedir el ultimo token, este debe ser "}" que indica el final de la gramática.

Si todo salió bien se retornará éxito en el análisis y proceder a aplicar otras operaciones a la gramática. Sino, en una área de mensajes se debe especificar cual es el tipo de error y sobre todo en que línea esta localizado, para así facilitarle al usuario la búsqueda de errores en su gramática.

3.4. Módulos de Analizador Sintáctico

El analizador sintáctico es un ensamble de diferentes módulos, aquellos serán en conjunto quienes brinden un correcto análisis. Sabemos que por comodidad, teoría o

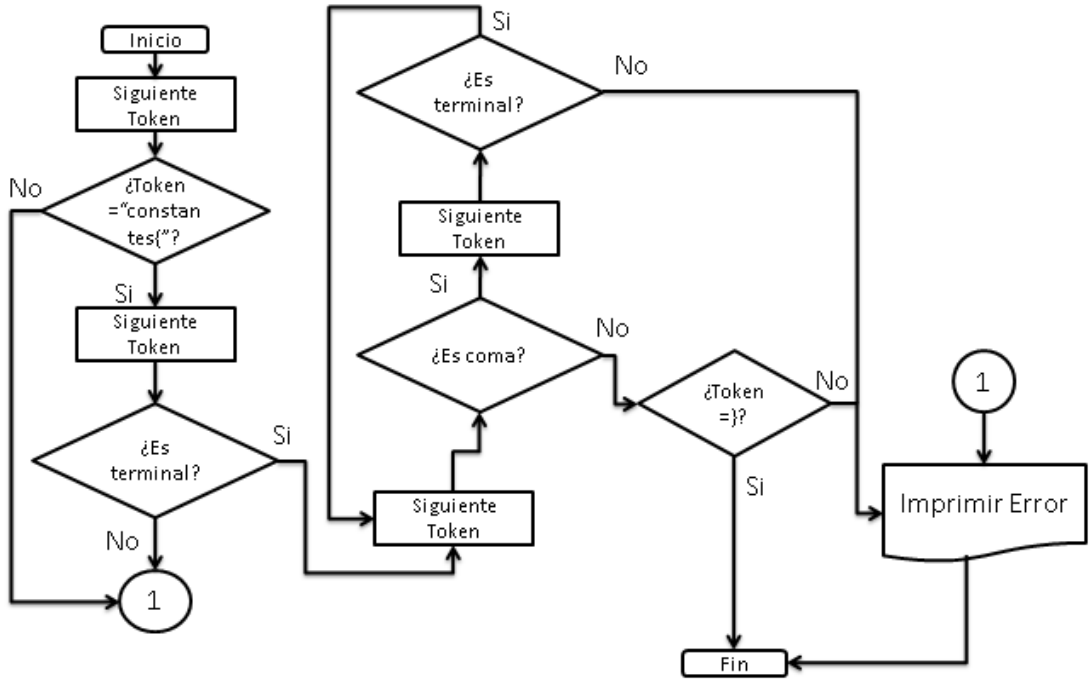


Figura 3.4: Diagrama para validar constantes.

experiencia, la división de grandes proyectos en módulos simplifican las tareas de tal forma que se hace más eficaz y comprensible el proceso de desarrollo. Anteriormente se ha dado la estructura completa del analizador sintáctico, ahora en esta sección se describe y desglosa explícitamente como funciona y se espera funcione cada uno de los módulos.

3.4.1. Validar constantes

Este módulo es el primero en ser visitado, consultado para revisar en el texto de gramática que se ha introducido la sección de constantes o terminales (nombrado así en formato de las gramáticas).

El objetivo de éste es revisar el orden de las constantes utilizando el diagrama de la figura 3.4, de manera inicial comienza por leer un token procedente del analizador sintáctico, se compara con la cadena "constantes", si es correcto, se solicita un segundo token el cual esperamos sea el símbolo "{", hasta aquí todo correcto.

Una vez especificada la inicialización de área de constantes se solicita otro token, el token devuelto no debe contener símbolos, únicamente palabras minúsculas o por lo menos la primera letra debe ser minúscula y el resto mayúsculas. Otra variante de tokens terminales validos son los que se encuentren encerrados entre comillas ("") ó paréntesis angulares (<>).

El mecanismo de solicitar primero que nada un terminal es para asegurarnos de que

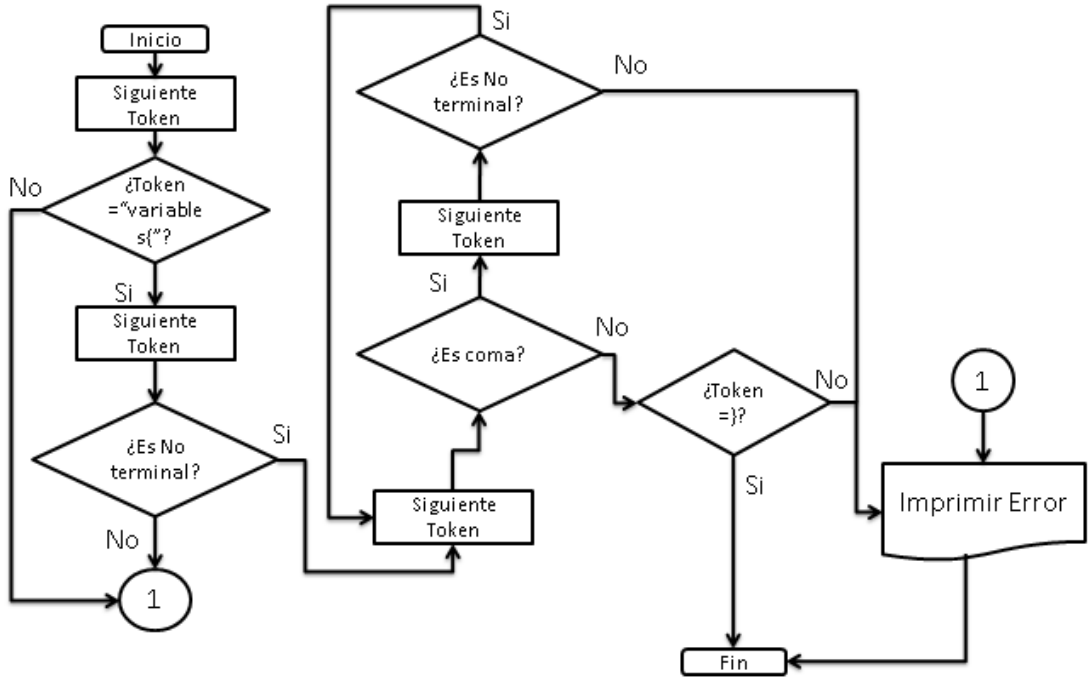


Figura 3.5: Diagrama para validar variables.

no se quede vacía el área de constantes, ya que esto podría traernos algunas dificultades en el resto de la definición de la gramática. Si no se obtuvo un token terminal será impreso un error de sintaxis, si se obtuvo entonces solicitamos otro token, si éste no es una coma esperamos que sea una llave de cierre "}" y así finalice la zona de definición de constantes o en caso contrario lanzar un error. Si es una coma procedemos a solicitar un siguiente token y evaluamos que sea un terminal, si no lo es imprimimos un error, de lo contrario volvemos al inicio del bucle donde se solicita un nuevo toquen y esperar que sea coma "," o llave de cierre "}" respectivamente según sea el caso.

3.4.2. Validar variables

El modulo de validación para variables o no terminales, muestra su diagrama en la figura 3.5. Goza de un esquema parecido al de validación de constantes con algunas diferencias, la principal diferencia se sitúa en la declaración de área de variables, en lugar de solicitar el token "constantes" seguido de una llave de apertura "{", ahora solicitará "variables" más una llave de apertura "{" y de ahí en fuera el lugar de terminales será sustituido por el de no terminales.

Cabe mencionar que los no Terminales o variables serán tokens compuestos de letras inicialmente con una mayúscula. Los módulos de identificación tanto de constantes como de variables tienen otra tarea aparte de la validación, la de añadir los tokens que se han validado y han sido aprobados a una tabla de reconocimiento. Para cuando lleguen al

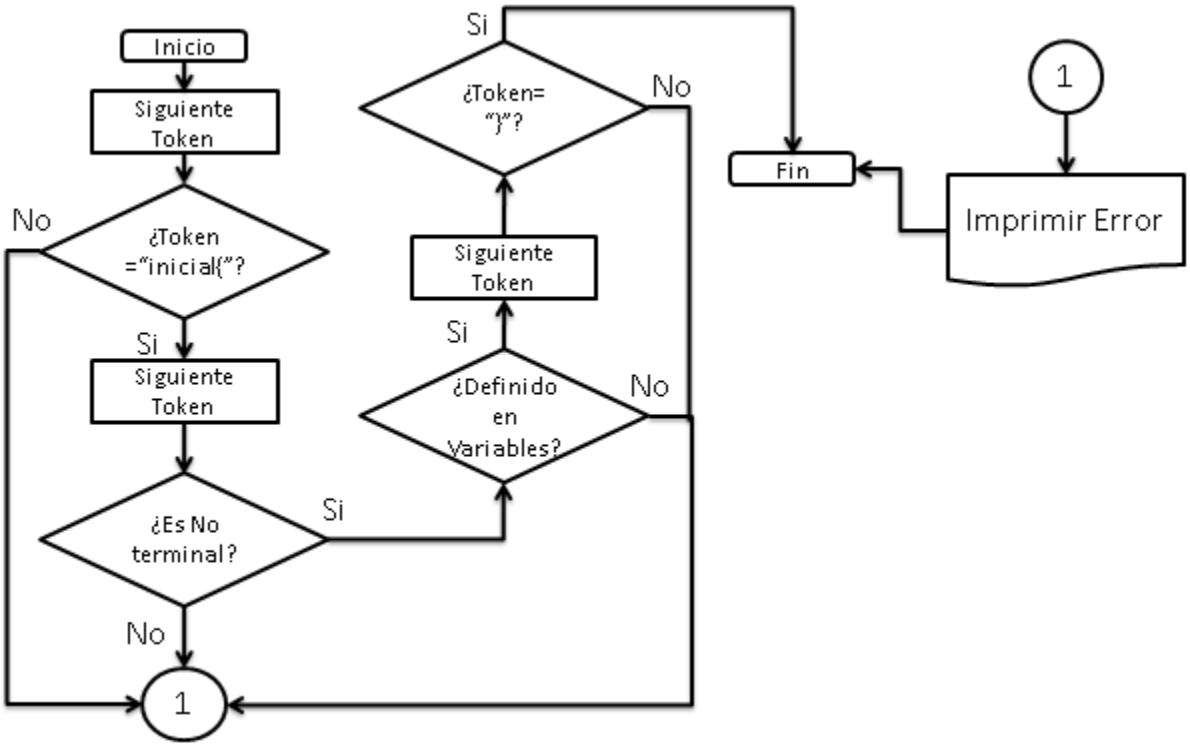


Figura 3.6: Diagrama para validar inicial.

modulo de validación de reglas, se analicen los tokens pero ya con referencia a la tabla creada en estos módulos.

3.4.3. Validar inicial

El modulo actual parece tener una labor un poco menos laboriosa, mostrado en la figura 3.6; se encarga únicamente de revisar que efectivamente el símbolo inicial que se introduce es valido y la forma de definirlo también sea la correcta.

Iniciemos con el mecanismo, primero que nada pide un token, al recibirlo lo compara con la cadena "inicial" y consecutivamente pide un siguiente token que deberá ser el símbolo de llave de apertura "{" de lo contrario mensaje de error. Después se solicita otro token, ahora éste debe ser un NoTerminal, pues las reglas de gramáticas establecen que ningún terminal o algún otro símbolo pueden tomar el lugar de símbolo inicial. Estando seguros de que contamos con un terminal el siguiente paso será revisar la tabla de Variables para constatar que efectivamente está registrado nuestro símbolo inicial.

Una vez comprobado lo anterior y si se ha cumplido finalmente solicitamos un siguiente token esperando sea la llave que cierra "}" para indicar el fin del área de definición de símbolo inicial para así continuar al módulo de verificación de reglas de producción.

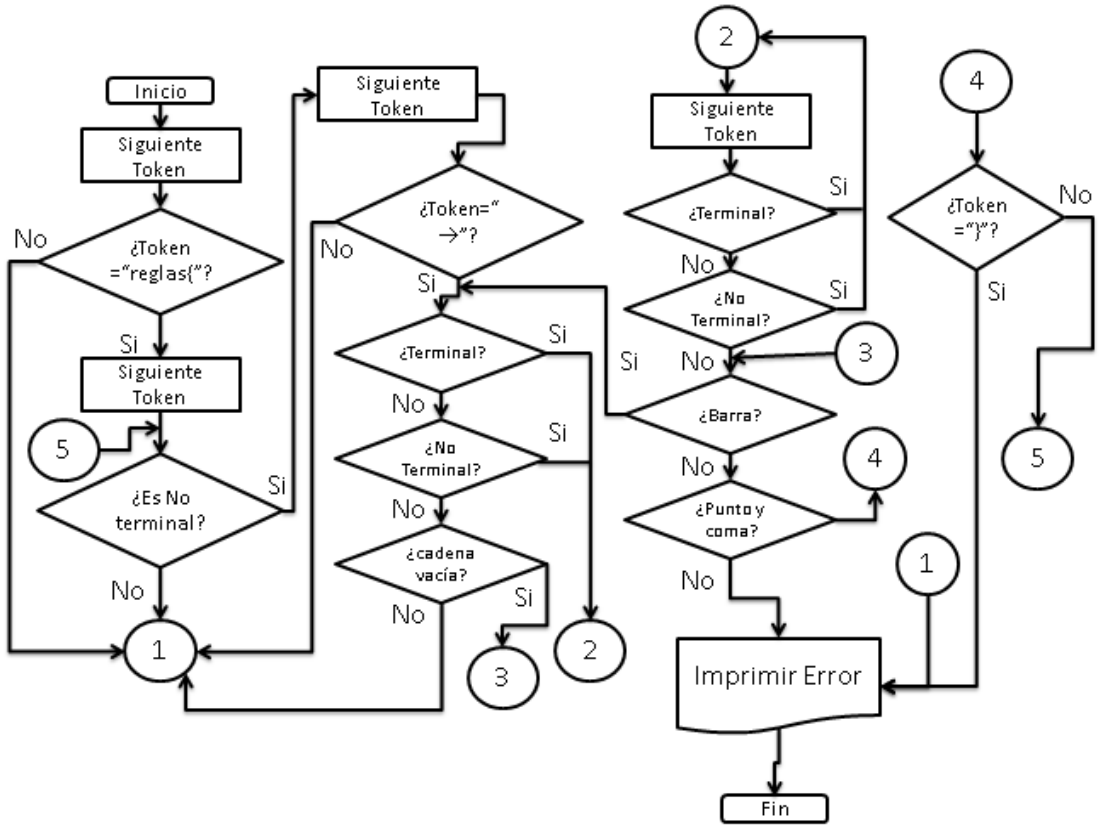


Figura 3.7: Diagrama para validar reglas de producción.

3.4.4. Validar reglas de producción.

Una vez descritos los módulos de análisis de variables y constantes, su adecuado funcionamiento además de agregar los respectivos en tablas de reconocimiento, viene un módulo esencial encargado de la revisión de las reglas de producción (la estructura de las reglas de producción se explica en el marco teórico), estas reglas tienen un encabezado, en la parte central las debe separar un símbolo "–>" (indica que produce el encabezado) y en la parte derecha finalmente esta el cuerpo de la producción (lo que sustituirá al encabezado).

Deben ser reconocidas y almacenadas para tener un mejor desempeño en el reconocimiento que se realizará más adelante con las gramáticas. El diagrama de la figura 3.7, esquematiza el flujo de los datos y su procesamiento. El mecanismo es como sigue; se inicia solicitando dos tokens un terminal y un símbolo para que sean compatibles con la cadena "reglas{", indicando que ha iniciado el área de declaración de reglas de producción; se solicita otro token, ¿Qué buscamos ahora?, pues el encabezado perteneciente a los no terminales o Variables, el tokén que se reciba debe ser un no terminal inmediatamente solicitamos otro token el cual debe ser "–>", mostrando así que el encabezado fue correcto y continuar con el análisis del cuerpo de la producción.

Quizá el análisis de los cuerpos de producción sea un poco más laborioso, pero para

eso el diagrama nos ilustra la forma que se encontró más conveniente. Una vez que se analizo el encabezado y el símbolo "−>" se solicita un siguiente token, nos topamos con que hay varias posibles opciones para el token que se obtenga. Por lo tanto buscando mayor comodidad se divide en dos bloques:

Bloque uno:

- Si token es un terminal: Revisamos que este registrado en la tabla de Constantes y si lo esta saltamos al bloque dos.
- Si token es No terminal: Revisamos que este registrado en la tabla de Variables y si lo esta saltamos al bloque dos.
- Si token es cadena vacía: Saltamos a la mitad del bloque dos. Como lo indica el diagrama 3.7. Esto con el fin de evitar que después de una cadena vacía se valide algún otro token, siendo que la cadena vacía debe estar sola.

Bloque dos: Al inicio del bloque dos, se solicita un siguiente token y después el que se obtiene, se compara con alguna de las siguientes opciones:

- Si token es terminal: Vuelve al inicio del bloque dos, ya que el token actual fue una constante valida y se dispone a validar otro token.
- Si token es No Terminal: Vuelve al inicio del bloque dos, ya que el token actual fue una Variable valida y se dispone a validar otro token.
- Si el token es una Barra "|": Indica que hay un separador de reglas, es decir otra regla pero con el mismo encabezado. Por lo tanto se inicia una nueva revisión de cuerpo de producción; regresando así al bloque uno.
- Si el token es Punto y Coma ";": Indica finalización de la regla de producción y consecuentemente se pueden analizar otras. Y ahora se espera que ocurra un token con llave que cierra "}", indicará la conclusión del área de declaración de reglas de producción. Sino, queda una opción, se trata de una nueva regla de producción y se debe analizar otra vez desde el encabezado de la regla. Es por eso que el enlace se vuelca hasta el conector cinco, donde comienza el análisis de una nueva regla.

En todos los casos anteriores, si sucede lo contrario o algo fuera de lo común al respecto del proceso, lanzara un error y detendrá el análisis. El error será impreso con su número de línea adjunto para mayor comodidad de su localización.

En las gramáticas existen claramente muchos tipos de ambigüedad y este editor no esta exento totalmente de ellas, en lo que cabe, funciona de manera correcta. Si los módulos han terminado con éxito su recorrido, el análisis fue un éxito. Y se puede proceder a aplicar otras operaciones sobre esa gramática pues se ha validado correctamente la sintaxis.

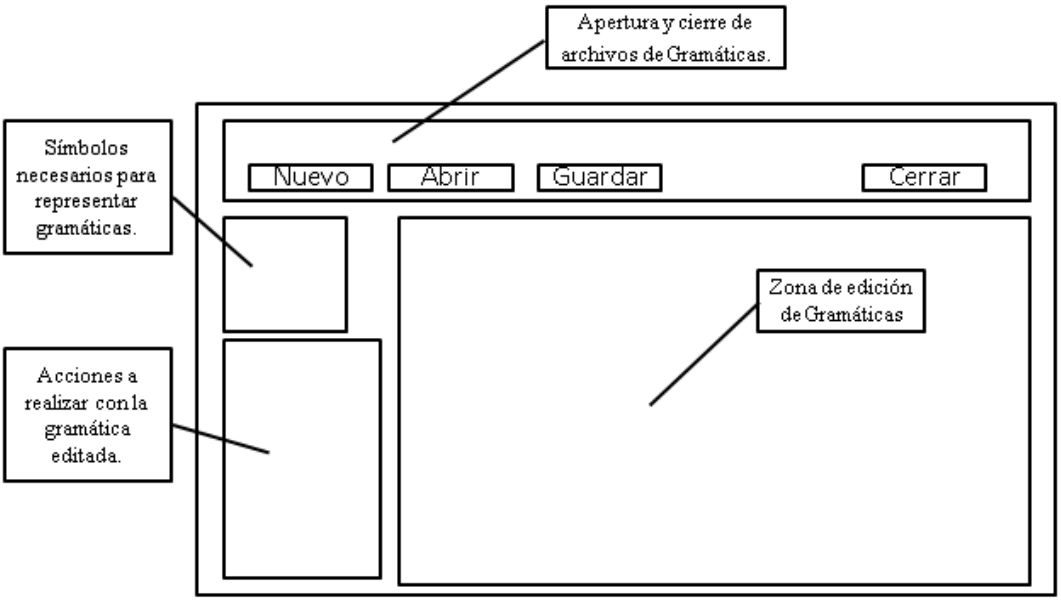


Figura 3.8: Esquema de la ventana principal.

3.5. Diseño de Interfaz

Como elemento principal del sistema existe la interfaz del usuario la cual obtendrá los datos necesarios para poder realizar las transacciones requeridas y posteriormente aplicar el análisis de gramáticas. La interacción para el usuario con el sistema debe ser cómoda respecto al manejo de datos y también mostrar facilidad de entendimiento referente a los resultados obtenidos .

3.5.1. Ventana principal

El esquema de la ventana principal de la figura 3.8, hace notar las áreas que son necesarias para editar una gramática como archivo de texto, por lo tanto contiene botones utilizados para la apertura y cierre de archivos así como la creación de nuevos, ésta similitud con otro tipo de aplicaciones hace más cómodo el uso de archivos.

De lado izquierdo notamos la existencia de un cuadro que contendrá símbolos necesarios para representar gramáticas. Pero!, ¿ Cuales son estos símbolos?. Pues basándonos en la representación básica GLC, los más utilizados serán "$\rightarrow$" (Separa encabezado de cuerpo en una producción), "ε" (cadena vacía), " $\mid$ " (Separación entre reglas), entre otros.

También de lado izquierdo pero en la parte inferior observamos un cuadro donde serán agregadas todas las acciones que se aplicaran o pueden ser aplicadas a la gramática en proceso. Algunos casos son el análisis Léxico y Sintáctico de la gramática, calculo de conjuntos predictivos, etc.

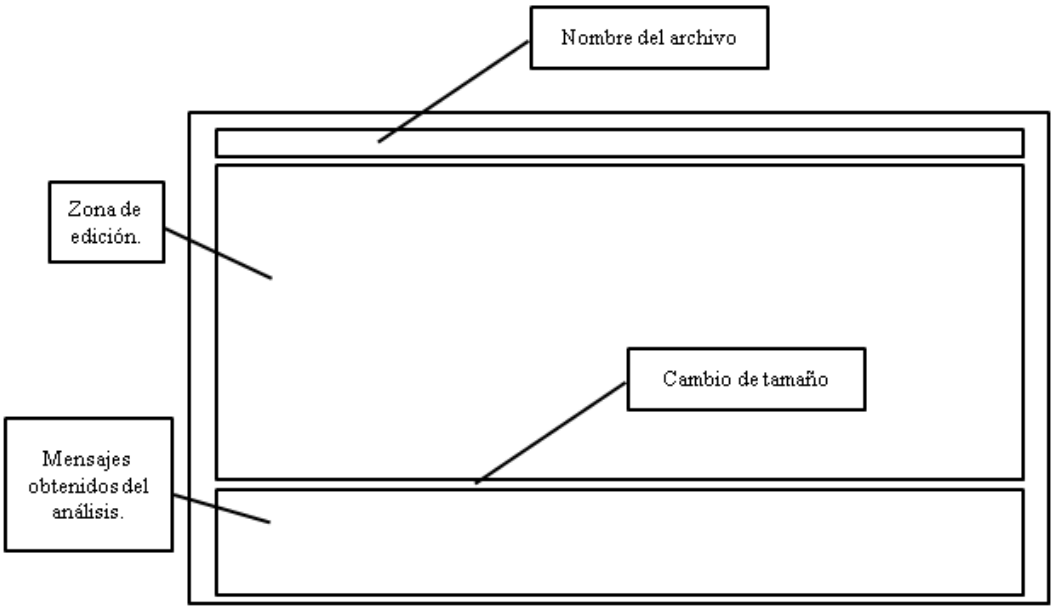


Figura 3.9: Esquema de edición.

Y en la parte Centro-Derecha se adjudica la edición de la gramática, introducción de texto, edición y corrección de errores, la zona de más interacción con el usuario en pocas palabras.

3.5.2. Frame de edición

El frame de edición que muestra el esquema de la figura 3.9, será la parte con la que debe existir una mayor interactividad entre usuario-aplicación. La barra superior de esta zona de edición enlista los nombres de archivos abiertos, cabe mencionar, no deben ser repetidos, pretendiendo que si se abre otra vez un archivo ya abierto con anterioridad, solo se posicione en el que se abrió primero. Con la finalidad de no acumular demasiados nombres y sobretodo repetidos.

Luego existe la parte en donde se deben introducir las gramáticas a manera de que tenga forma de un compilador.

Más abajo existe una pequeña barra que permite el desplazamiento o alargamiento entre la ventana de mensajes y la de introducción de texto.

Y por ultimo existe la zona de mensajes, que tendrá encomendada la función de mostrarnos los errores léxicos-sintácticos así como otros avisos que surjan según la gramática en cuestión.

3.5.3. Ventanas de trabajo

El esquema de las ventanas de trabajo comunes de la figura 3.10. Tendrán principalmente esa forma o cumplirán con esa expectativa de trabajo, contando como

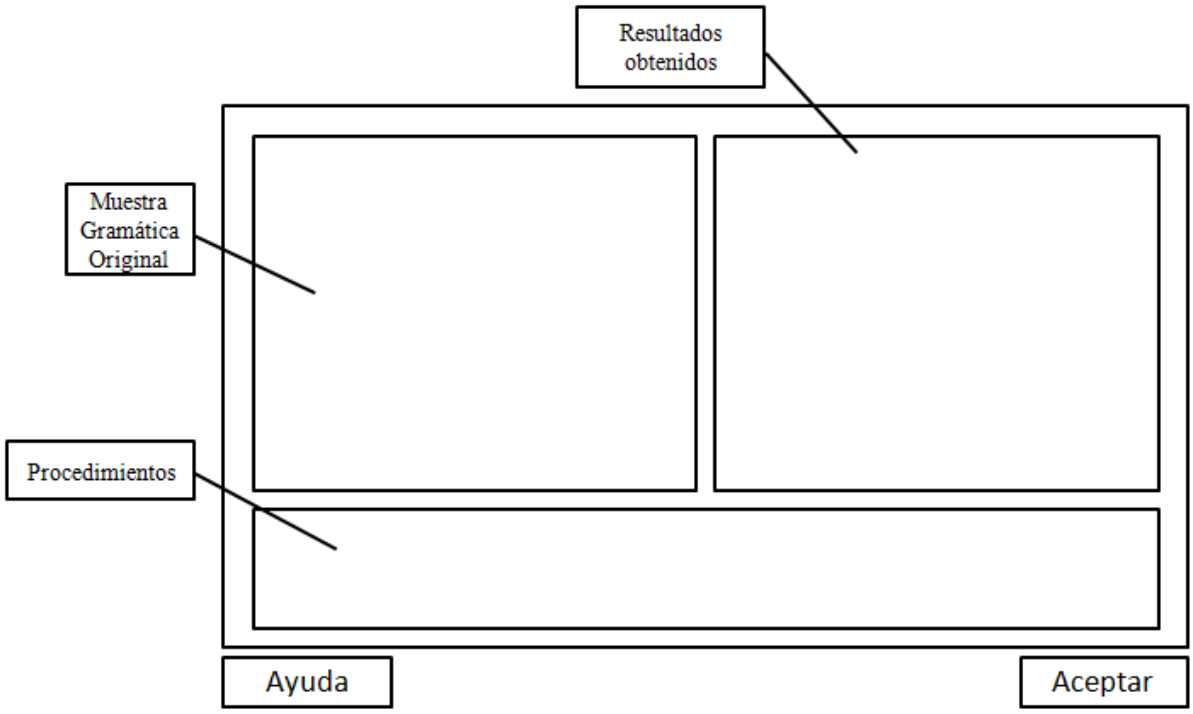


Figura 3.10: Esquema de ventanas de trabajo comunes.

base con tres secciones indispensables. El área que muestra la gramática de origen con la que se está trabajando actualmente. El área de Resultados obtenidos donde se mostraran principalmente conclusiones de los cálculos realizados. Estos datos podrían ser requeridos para futuras operaciones. Finalmente el área de procedimientos encargada de mostrar el proceso que se ha llevado a cabo para llegar a los resultados mostrados. Ésta con el fin de ilustrar y mejorar la comprensión de aquellos que deseen aprender la correcta forma y aplicación de las reglas correspondientes, según el evento en cuestión. En la parte inferior notamos la presencia de dos botones. "Ayuda", encargado de lanzar una ventana en la cual se especifica claramente las reglas que se deben aplicar en cada proceso. Y "Aceptar", simplemente para salir de esa ventana de trabajo actual.

Capítulo 4

Implementación y Pruebas

En este capítulo entramos en la fase de implementación de algoritmos descritos en capítulos anteriores, aquí es donde se define y establece realmente si la relación entre la planeación del algoritmo y la implementación fueron correctas. Serán mostradas a continuación las ventanas y su respectiva descripción sobre su significado, estarán acompañadas de un pequeño ejemplo para hacer más ilustrativo el correcto funcionamiento de la aplicación.

4.1. Área principal

La primera ventana muestra la estructura de la aplicación en general, se muestra en la figura 4.1. Tal y como se describió en el esquema de diseño, será el punto clave principal con el que se encontrará el usuario y con el que de cierta manera tendrá más interacción al editar sus gramáticas. En la parte superior se trabaja lo que viene siendo apertura, salvamento y cierre de gramáticas. A los laterales se localizan botones de inserción de símbolos esenciales para la creación de gramáticas y más abajo las correcciones que se le pueden aplicar a la gramática tanto para mejorarla como para verificar que sea del tipo LL(1), que en nuestro caso será indispensable.

En la ventana de edición se muestra un área donde es posible trabajar cómodamente con el texto, se observa el trabajo del analizador léxico y sintáctico los cuales trabajan en dúo y hacen ver que existe recursividad a la izquierda, además la gramática necesita factorización

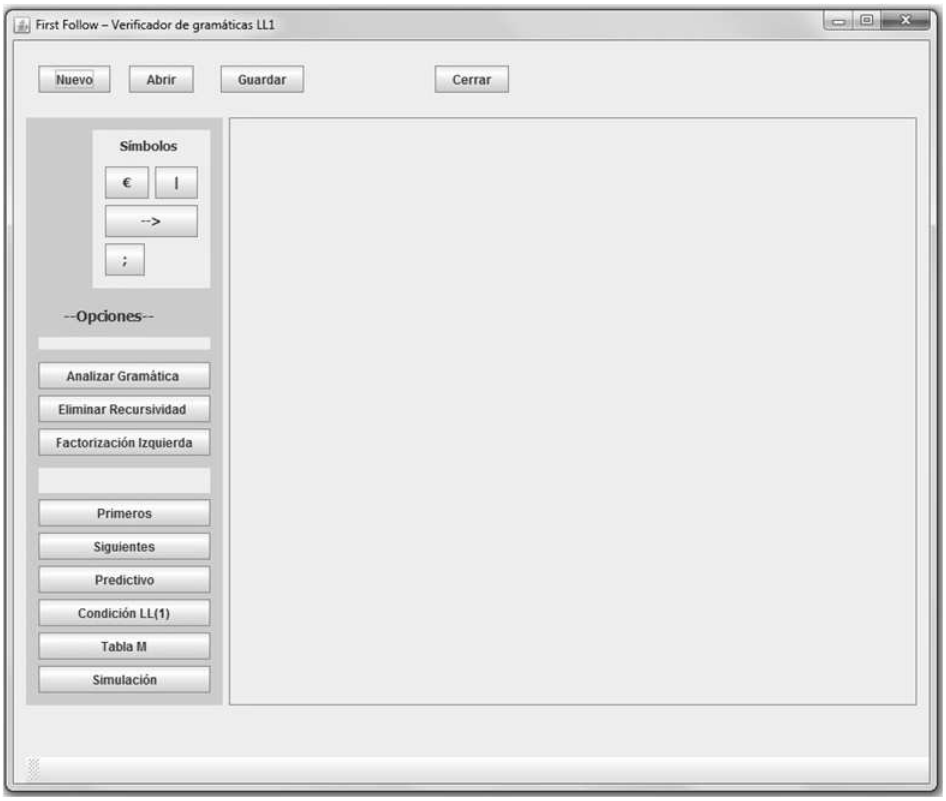


Figura 4.1: Ventana principal.

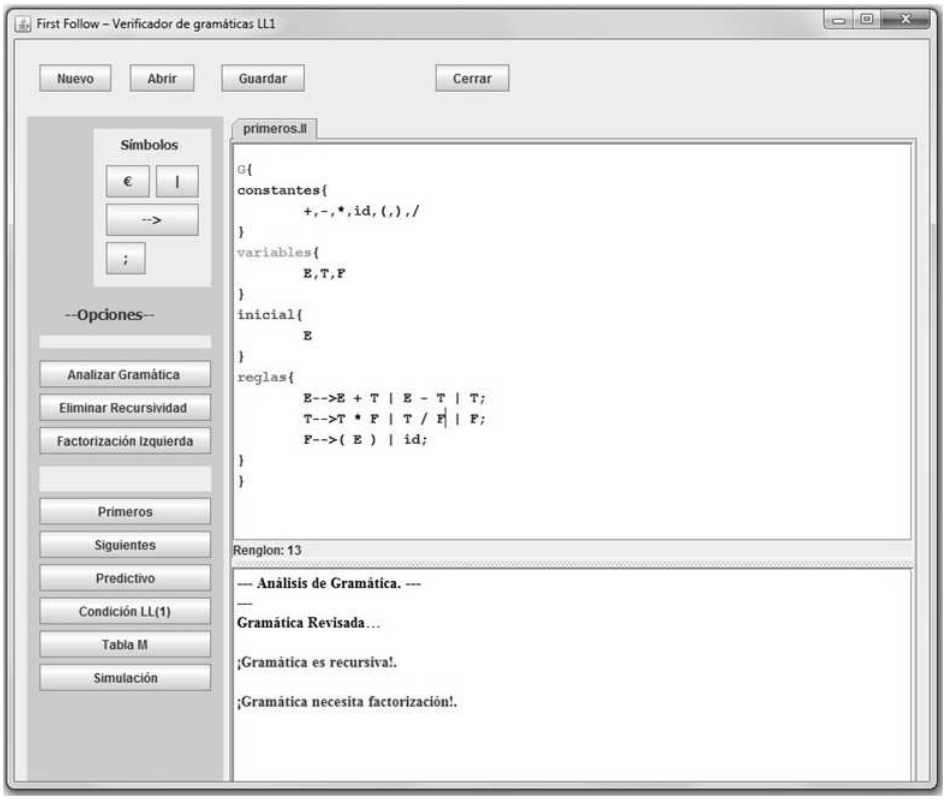


Figura 4.2: Ventana edición .

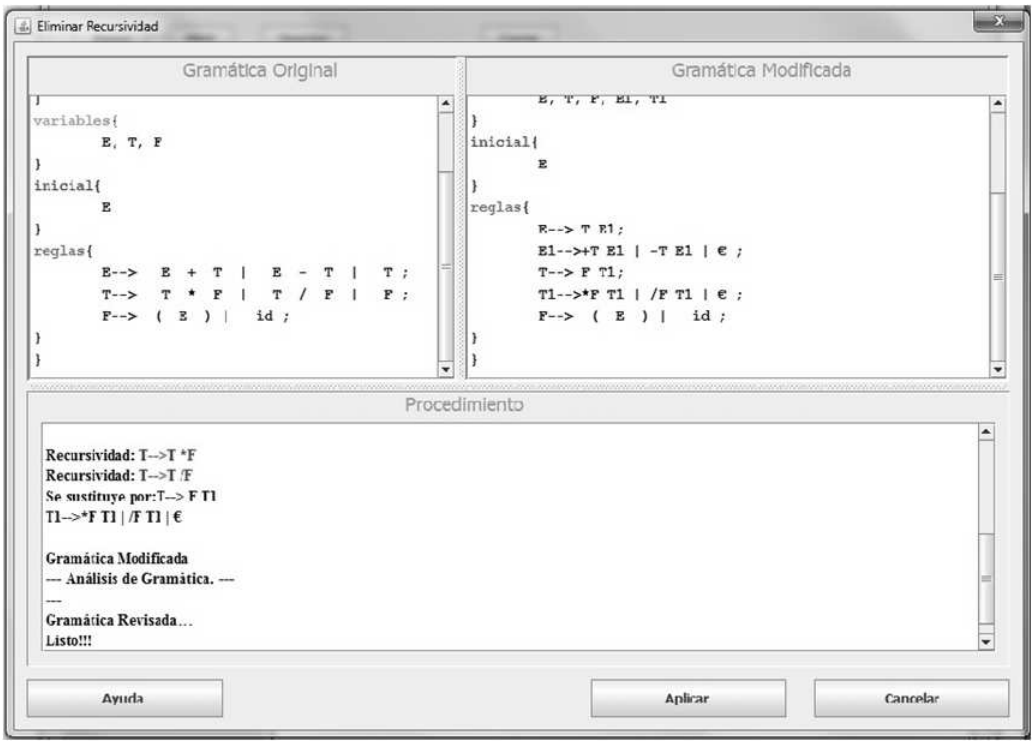


Figura 4.3: Ventana eliminar recursividad..

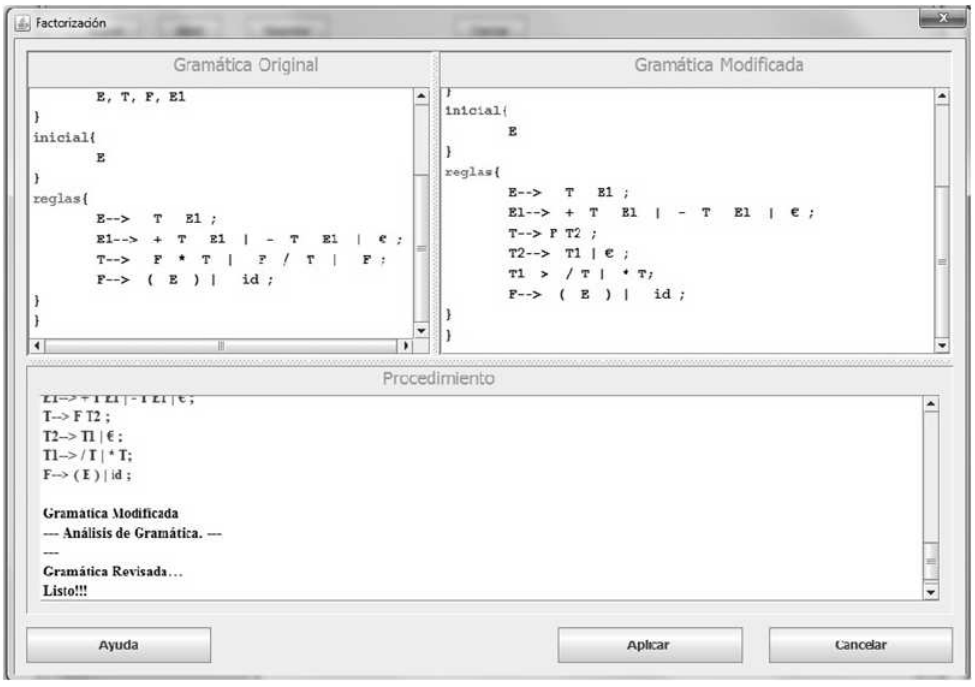


Figura 4.4: Ventana Factorización.

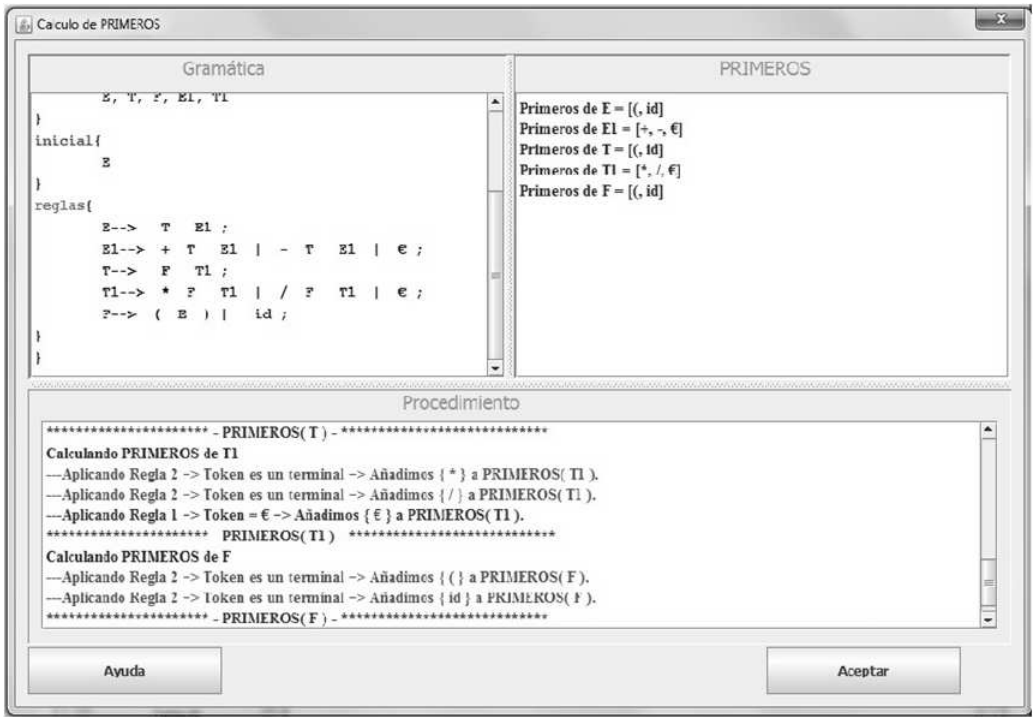


Figura 4.5: Ventana calculo de PRIMEROS.

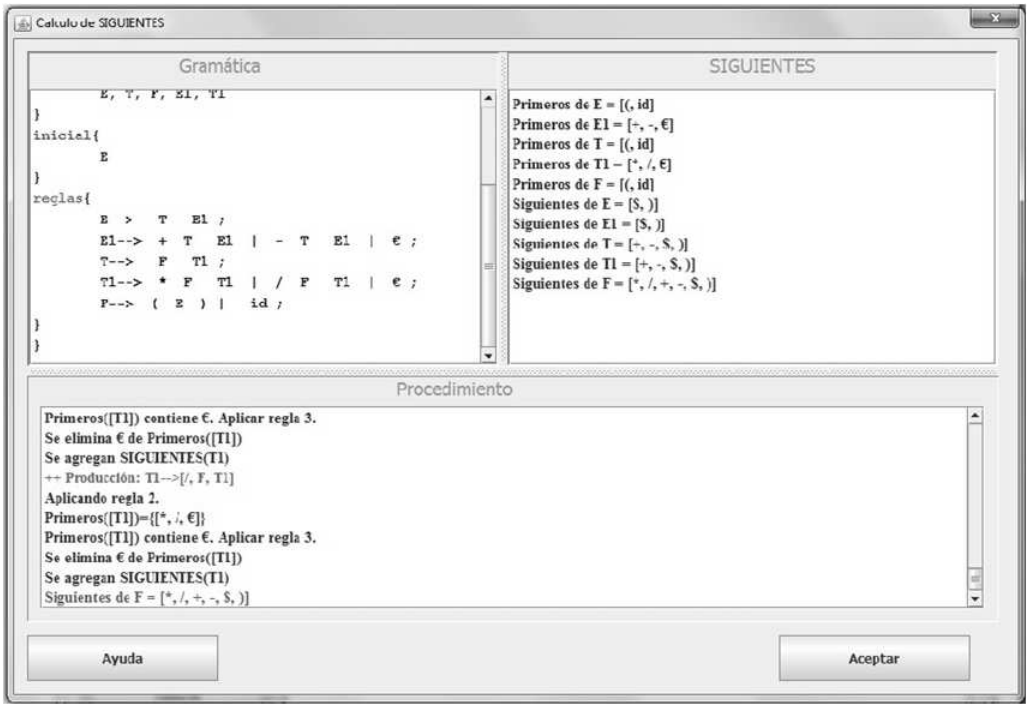


Figura 4.6: Ventana calculo de SIGUIENTES.

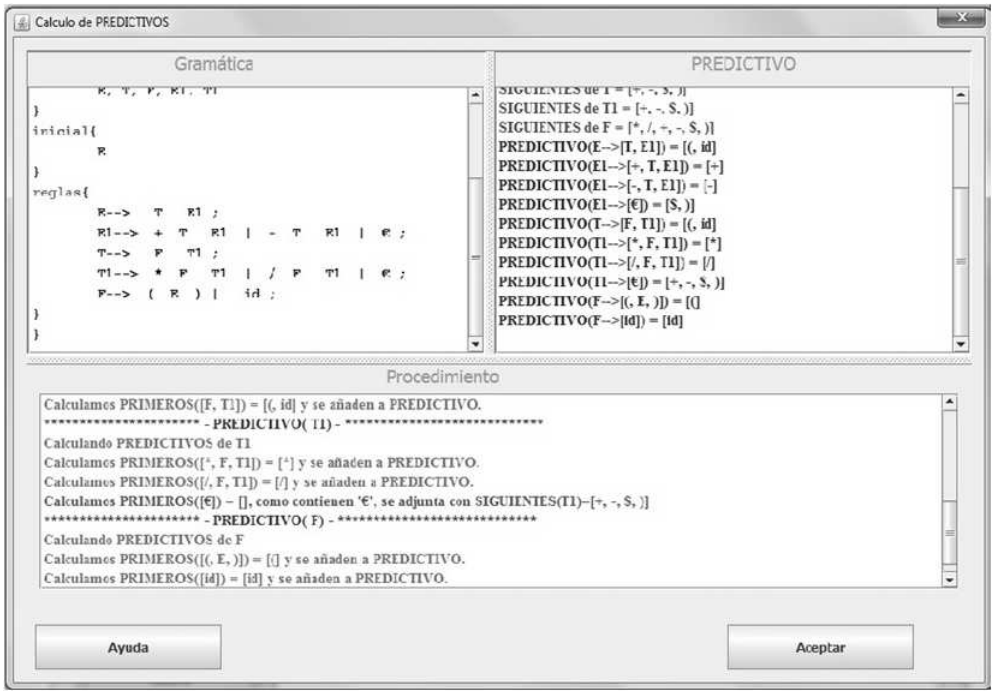
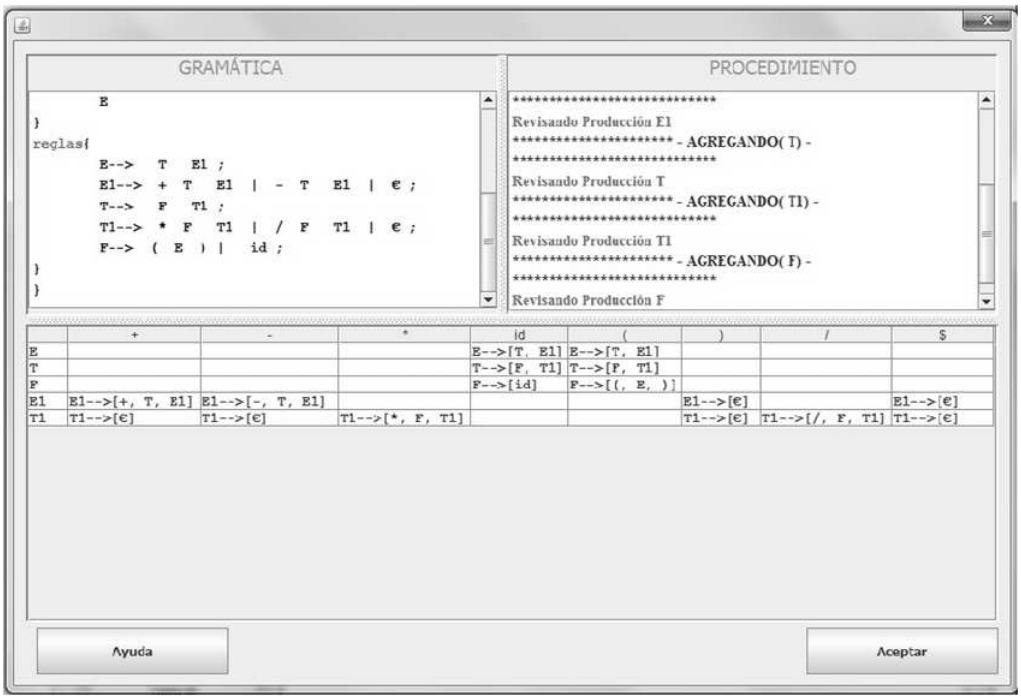


Figura 4.7: Ventana calculo de PREDICTIVOS.



	+	-	*	id	(	)	/	$
E				E-->[T, E1]	E-->[T, E1]			
T				T-->[F, T1]	T-->[F, T1]			
F				F-->[id]	F-->[(, E,)]			
E1	E1-->[+, T, E1]	E1-->[-, T, E1]				E1-->[€]		E1-->[€]
T1	T1-->[€]	T1-->[€]	T1-->[*, F, T1]			T1-->[€]	T1-->[/, F, T1]	T1-->[€]

Figura 4.8: Ventana Tabla M.

Bibliografía

[1] V. Aho Alfred. *"Compiladores – principios, técnicas y herramientas"*. Segunda Edición. Pearson Educacion, México, 2008.

[2] *"Notas para el curso de compiladores"*. Facultad de Ciencias de la Computación. BUAP. México, Primavera 2010.

[3] Godoy Guillermo. *"Apuntes de Compiladores"*. Facultad de informática de Barcelona. Barcelona, 2007.

[4] Brena Ramón. *"Autómatas y Lenguajes – Un enfoque de diseño – "*. Tecnológico de Monterrey. México, Verano 2003.