Compiladores
Facultad de Ciencias UNAM
Análisis Sintáctico y Gramáticas Libres de Contexto

Lourdes Del Carmen González Huesca [*]

8 de marzo de 2024

La segunda fase del compilador y del front-end es considerada la parte más importante de la compilación, esta es el análisis sintáctico o *parsing.* [1]

Los lenguajes de programación son definidos formalmente a través de gramáticas libres de contexto. El objetivo de esta fase es reconocer palabras y verificar que la secuencia de tokens pueda ser generada por la gramática del lenguaje.

Durante esta fase se utiliza el resultado de la fase anterior, es decir se utiliza la secuencia de tokens creada por el analizador léxico. Se toma el nombre o el tipo de los tokens generados por el lexer para crear una derivación de la secuencia de tokens para asegurar que el programa fuente coincide con la estructura gramatical del lenguaje. Adicionalmente a la derivación se puede generar una estructura concreta que represente gráficamente la forma gramatical del programa, a esta estructura se le llama árbol de sintaxis concreta o ***parse tree***.

El análisis de la cadena de tokens siempre se realiza de izquierda a derecha y revisando símbolos por adelantado para construir el árbol de sintaxis concreta. El análisis del programa se lleva a cabo seleccionando una regla de producción de la gramática que coincida con el símbolo de lectura en turno y haciendo coincidir la producción con los símbolos subsecuentes. La representación del programa generada por el *parser* es un árbol de sintaxis concreta o *parse tree* cuyos nodos son símbolos no-terminales de la gramática y las hojas son símbolos terminales [2]. Si el programa no es reconocido como una secuencia correcta de palabras, según la gramática que define al lenguaje, entonces se deben reportar los errores sintácticos.

Gramáticas libres de contexto

Las gramáticas libres de contexto definen lenguajes al establecer las reglas para estructurar palabras. Estas gramáticas están en correspondencia con los lenguajes libres de contexto y los autómatas de pila según la jerarquía de Chomsky para lenguajes. El lenguaje generado por una gramática se define como todas aquellas derivaciones posibles o todos los árboles de derivación posibles partiendo de dicha gramática. Lo anterior debe excluir a gramáticas que permitan dos o más derivaciones para una misma palabra. Entonces las gramáticas a considerar son aquellas que no son recursivas, no tienen transiciones épsilon y lo más importante: no son ambiguas.

[*]Material revisado por el servicio social de Apoyo a la Docencia y Asesoría Académica en la Facultad de Ciencias de la UNAM con clave 2023-12/12-292, de Diana Laura Nicolás Pavia.

[1]Cuya traducción al español es simplemente la acción de analizar sintácticamente un enunciado.

[2]A diferencia de un árbol de sintaxis abstracta o *abstract syntax tree* cuyos nodos y hojas son símbolos terminales.

Definición 1. Una gramática se dice **ambigua** si existe una palabra w con dos o más árboles de derivación distintos. En general una palabra puede tener más de una derivación, pero un sólo árbol y en tal caso no hay ambigüedad.
Un lenguaje L es ambiguo si existe una gramática ambigua G que genera a L y decimos que un lenguaje es **inherentemente** ambiguo si *todas* las gramáticas que lo generan son ambiguas.

Para eliminar la ambigüedad de una gramática no existe un algoritmo pero hay algunas maneras de reformular las reglas de producción de la gramática para eliminarla o minimizarla, por ejemplo se puede considerar la precedencia de algunos símbolos o el órden en que se pueden asociar algunos símbolos. La ambigüedad es sinónimo de no-determinismo, es decir que al procesar una cadena de símbolos se pueden obtener dos o más árboles. Esto se traduce en una indecisón sobre la estructura o forma del programa a analizar, lo cual es inaceptable.

Consideraremos las transformaciones de gramáticas que eliminan tanto ε-producciones y la recursión izquierda.

Definición 2. Una variable A se llama **anulable** si $A \to^\star \varepsilon$, es decir si una derivación que empieza en A genera la cadena vacía.
Se da un algoritmo para hallar variables anulables:

Iniciar el conjunto *Anul* con las variables que tienen ε como producción

$$Anul := \{A \in V \mid A \to \varepsilon \ \in P\}$$

Repetir la incorporación de variables que tienen producciones cadenas de variables anulables

$$Anul := Anul \cup \{A \in V \mid \exists\ A \to w \ \in P,\ \ w \in Anul^\star\}$$

Hasta que no se añaden nuevas variables a *Anul*

Una vez que se han identificado las variables anulables, la siguiente transformación de una gramática libre de contexto elimina exactamente las $\square$-producciones:

Para cada producción en la gramática que tenga la forma $A \to w_1 \dots w_n$ se deben agregar las producciones $A \to v_1 \dots v_n$ que son resultantes de los cambios de símbolos donde:

- $v_i = w_i$ si $w_i \notin Anul$, se respetan las variables no anulables
- $v_i = w_i$ ó $v_i = \varepsilon$ si $w_i \in Anul$, las variables anulables pueden dejarse o eliminarse

Verificando que no se anulen todos los v_i al mismo tiempo.

Es decir, se van a respetar las producciones existentes y si alguna contiene las variables anulables se agregarán las producciones que resulten de eliminar las variables anulables en esa producción. Las ε-producciones desaparecerán.

Definición 3. Una producción recursiva por la izquierda es de la forma $A \to Aw$ con $w \in (V \cup T)^*$.
Ellas serán reemplazadas por producciones cuya recursión es por la derecha de la siguiente forma, para cada variable o símbolo no-terminal A crear dos categorías de reglas:

1. $A_{izq} = \{A \to Au_i \in P \mid u_i \in (V \cup T)^*\}$
2. $A_{der} = \{A \to v_j \in P \mid v_j \in (V \cup T)^*\}$

A continuación se obtendrán las reglas para la variable A como sigue:

$$A \to v_1 \mid \cdots \mid v_m \mid v_1 Z \mid \cdots \mid v_n Z$$

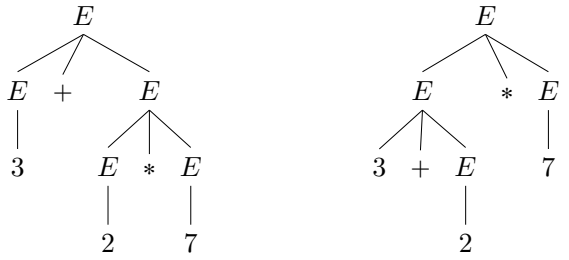
Además se incorporarán las siguientes reglas a P:

$$Z \to u_1 Z \mid \cdots \mid u_n Z \mid u_1 \mid \cdots \mid u_n$$

Ejemplo 1 (Gramática para expresiones aritméticas)**.** Consideremos la gramática

$$E \to E + E \mid E - E \mid E * E \mid E/E \mid (E) \mid n$$

donde n son los números naturales. Esta gramática es ambigüa ya que se pueden obtener diferentes árboles de sintaxis concreta para una misma cadena, por ejemplo:



Se puede transformar la gramática anterior en gramáticas equivalentes que no sean ambigüas [3]:

1. Gramática recursiva por la izquierda no-ambigüa

$$\begin{aligned} E &\to E + T \mid E - T \mid T \\ T &\to T * F \mid T/F \mid F \\ F &\to (E) \mid n \end{aligned}$$

2. Eliminación de la recursión por la izquierda

$$\begin{aligned} E &\to TE' \\ E' &\to +TE' \mid -TE' \mid +T \mid -T \\ T &\to FT' \mid F \\ T' &\to *FT' \mid /FT' \mid *F \mid /F \\ F &\to (E) \mid n \end{aligned}$$

Observaciones Para cada gramática libre de contexto existe un parser que toma $O(n^3)$ en analizar una cadena de n símbolos terminales (eg. algoritmo CYK). Para lenguajes de programación se pueden obtener analizadores más rápidos y en la práctica es posible obtener uno que tome tiempo lineal.

Parsers

Los métodos de análisis, generalmente se clasifican en dos:

- Analizadores *top-down* que construyen el árbol desde la raíz y hacia las hojas
- Analizadores *bottom-up* que construyen el árbol desde las hojas hacia la raíz

[3] Se pueden consultar referencias de cursos de Autómatas y Lenguajes Formales para revisar métodos para transformar gramáticas libres de contexto.

aunque también existen los analizadores o parsers universales para cualquier tipo de gramática en forma normal de Chomsky implementando el algoritmo CYK (Cocke–Younger–Kasami) (bottom-up y dinámico).

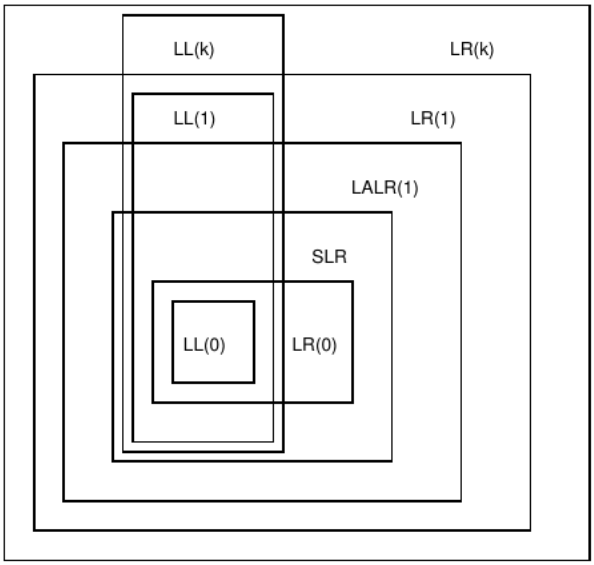
Los tipos de parsers están en función del tipo de gramática libre de contexto que describa al lenguaje de programación. Las formas normales ayudan a obtener gramáticas cuyas producciones sean uniformes.

Definición 4 (Gramática **LL**)**.** Decimos que una gramática pertenece a la clase LL si para cualquier símbolo no terminal a la izquierda de las producciones con más de dos reglas, los conjuntos de sus derivaciones son disjuntas, es decir para cada $A \to \alpha \mid \beta$ sucede que $\textsc{First}(\alpha) \cap \textsc{First}(\beta) = \varnothing$.

En otras palabras, una gramática tipo **LL** tiene la característica de que para cada símbolo no terminal A, una derivación $S \to^\star w_1 A\alpha \to w_1 v\alpha \to^\star w_1 w_2 w_3$ coincide con otra derivación con el mismo prefijo hasta la reescritura de A, i.e. las producciones de A sólo existe una regla que lo permite.

Este tipo de gramáticas son lineales, característica que da una transformación a una forma normal (Chomsky o Greibach), además no tienen recursión por la izquierda. Un parser **LL(k)** es uno para una gramática de este estilo y donde se revisan k tokens por adelantado. Se creará un parse tree top-down y en preorden con los tokens adelantados para crear un prefijo útil. Una gramática **LL** sin transiciones épsilon es una gramática **SLR**. Las gramáticas **LR** son también lineales y son más expresivas, permiten recursión por la izquierda.

Las gramáticas libres de contexto se pueden clasificar mediante el siguiente diagrama [4]:



Funciones auxiliares

Se definen a continuación dos funciones auxiliares a usarse en los algoritmos de los parsers. Estas funciones consideran los símbolos terminales y no-terminales del lenguaje y permiten escoger una producción al considerar los siguientes símbolos en la cadena de entrada.

Definición 5 (Función First)**.** Esta función calcula el conjunto de símbolos terminales que están al inicio de las palabras derivadas de una cadena:

$$\textsc{First}(\alpha) = \{a \in \Sigma \mid \exists w, \alpha \to^* aw\}$$

Recursivamente para un símbolo se tiene:

[4] Imagen inspirada de [**?**] y tomada de [**?**].

- Si $X \in \Sigma$ entonces $\textsc{First}(X) = \{X\}$.
- Si X es no-terminal y $X \to Y_1 Y_2 \dots Y_k$ es una producción con $k \geqslant 1$, entonces si $a \in \textsc{First}(Y_1)$ se tiene $a \in \textsc{First}(X)$.
  Es decir que $\textsc{First}(Y_1) \subseteq \textsc{First}(X)$.

Esta definición se puede generalizar a una cadena.

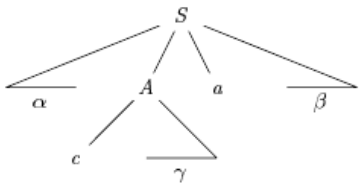
Definición 6 (Función Follow)**.** La función Follow para un símbolo no-terminal X calcula el conjunto de símbolos terminales que aparecen en una derivación cualquiera justo despúes de X:

$$\textsc{Follow}(X) = \{a \in \Sigma \mid A \to vXaw,\ v, w \in \Gamma \cup \Sigma\}$$

Para esta función se agrega un nuevo símbolo para indicar el final del archivo a procesar mediante #. Este símbolo será útil en el analizador. Recursivamente:

- $\# \in \textsc{Follow}(S)$ con S el (nuevo) símbolo inicial de la gramática.
- Si $A \to uXw$ entonces todo símbolo en $\textsc{First}(w)$ está en $\textsc{Follow}(X)$ excepto ε.
- Si existen producciones $A \to \alpha X$ o, $A \to \alpha X \beta$ donde $\varepsilon \in \textsc{First}(\beta)$, entonces $\textsc{Follow}(A) \subseteq \textsc{Follow}(X)$.

Estas funciones también ayudarán a identificar el tipo de gramática [**?**].
Veamos gráficamente cómo se ven los símbolos de estas funciones en un árbol de sintaxis [5]:



Terminal c is in $\textsc{first}(A)$ and a is in $\textsc{follow}(A)$

Ejemplo 2 (Cálculo de los conjuntos First y Follow)**.** Consideremos la grámática para expresiones aritméticas

$$\begin{array}{lcl} E & \to & TE' \\ E' & \to & +TE' \mid -TE' \mid +T \mid -T \\ T & \to & FT' \mid F \\ T' & \to & *FT' \mid /FT' \mid *F \mid /F \\ F & \to & (E) \mid n \end{array}$$

Los resultados de las funciones auxiliares son:

	E	E'	T	T'	F	S
First	$\{(, n\}$	$\{+, -\}$	$\{(, n\}$	$\{*, /\}$	$\{(, n\}$	$-$
Follow	$\{\#,)\}$	$\{\#,)\}$	$\{+, -, \#,)\}$	$\{+, -, \#,)\}$	$\{*, /, +, -, \#,)\}$	$\{\#\}$

[5] Imagen tomada de [**?**].

Ejercicios

1. Describe de manera clara y concisa, en tus palabras, el objetivo y uso del analizador léxico en el proceso de compilación

2. Considere la siguiente gramática, donde x se usa para identificadores *id* y n para números *num*.

$$E \rightarrow n \mid x \mid lam\ x.E \mid E\ E \mid (E) \mid E \oplus E$$

   a) Esta gramática es ambigua. De dos ejemplos de expresiones que tengan diferentes derivaciones para mostrar la ambiguedad de la gramática.

   b) Resuelva la ambiguedad mediante criterios que usted mismo proponga, es decir información adicional o algunas convenciones. Use ejemplos para justificar sus decisiones.

   c) Transforme la gramática dada en una que no sea ambigua y que acepte el mismo lenguaje. Esta nueva gramática debe respetar los criterios dados en el inciso anterior.

3. Dado el lenguaje $\mathbb{C}^2$ definido como

$$\mathbb{C}^2 := v \mid v\ op\ v$$

   Donde

   - $id :=$ Cualquier cadena alfanumérica
   - $elem := \mathbb{R} \mid id$
   - $v := (elem \oplus elem, elem \oplus elem) \mid id$
   - $op := + \mid - \mid *$

   Realiza lo siguiente

   - Da una expresión regular para cada uno de los elementos del lenguaje
   - Define cada uno de los tokens que se van a generar en el análisis léxico (Definiendo claramente cuál es su nombre y el valor que guardará)
   - Di claramente qué hará el Scanner dado el lenguaje que tenemos
   - Diseña los autómatas encargados de regresar los tokens
   - Dada tu implementación anterior muestra los tokens (o bien error) que se generarán con las siguientes expresiones.
     - $var1 + (3 \oplus 53)$
     - $(+, Var \oplus 3)$
     - $(0 \oplus 54, 6 \oplus yupl342) * ($

4. Diseña un archivo para ser usado por el generador automático flex con reglas para reconocer dígitos, identificadores, la palabra reservada while, paréntesis, contabilizar el número de líneas del archivo fuente e ignorar comentarios y espacios en blanco.

   Consideraciones:

   - Los identificadores son cadenas alfanuméricas que inician en carácter alfabético.
   - Los comentarios no pueden contener saltos de líneas, su formato es el siguiente:

$$/*\ Comentario\ */$$