Introducción

En la clase de hoy, empezaremos mostrando los teoremas de equivalencia entre autómatas y gramáticas.

A continuación, mostraremos un algoritmo fundamental en el diseño de compiladores, el cual nos permite construir un autómata con pila equivalente a una gramática incontextual.

Después de eso, estudiaremos el tipo de gramáticas incontextuales que se utiliza en el diseño de compiladores. Dicho intuitivamente, para poder diseñar compiladores necesitamos utilizar gramáticas que determinen de forma única la estructura de las palabras que generan. A dichas gramáticas incontextuales se las llama gramáticas no ambiguas.

Empezamos recordando el concepto de gramática incontextual.

Gramáticas incontextuales

Una gramática incontextual es una estructura $G = (V, \Sigma, P, S)$ donde:

(1) V es un alfabeto, a cuyos símbolos se les llama variables.

(2) Σ es un alfabeto disjunto de V (es decir, $V \cap \Sigma = \emptyset$), a cuyos símbolos se les llama terminales.

(3) P es un subconjunto finito de $V \times (V \cup \Sigma)^*$, a cuyos elementos se les llama producciones (o reglas).

(4) $S \in V$ es la variable inicial.

Y decimos que una gramática $G = (V, \Sigma, P, S)$ es regular, si toda producción de P es de la forma $A \longrightarrow xB$ o $A \longrightarrow x$ donde $A, B \in V$ y $x \in \Sigma^*$.

Teorema de equivalencia de autómatas finitos y gramáticas regulares

Este teorema afirma que para todo lenguaje L, existe un autómata finito M tal que $L(M) = L$ si y sólo si existe una gramática regular G tal que $L(G) = L$.

El teorema anterior expresa, por tanto, la equivalencia entre las gramáticas regulares y los autómatas finitos.

Teorema de equivalencia de autómatas con pila y gramáticas incontextuales

Este teorema afirma que para todo lenguaje L, existe un autómata con pila M tal que $L(M) = L$ si y sólo si existe una gramática incontextual G tal que $L(G) = L$.

El teorema anterior expresa, por tanto, la equivalencia entre las gramáticas incontextuales y los autómatas con pila.

La demostración del anterior teorema nos proporciona además el siguiente algoritmo central en el diseño de compiladores.

Algoritmo para construir un autómata con pila equivalente a una gramática incontextual.

Si $G = (V, \Sigma, P, S)$ es una gramática incontextual, definimos el autómata con pila $M = (\{q_0, f\}, \Sigma, V \cup \Sigma, \Delta, q_0, \{f\})$ donde Δ consta de las siguientes transiciones:

(1) $((q_0, \lambda, \lambda), (f, S))$.

(2) $((f, \lambda, A), (f, x))$ para cada producción $A \longrightarrow x$ de P.

(3) $((f, a, a), (f, \lambda))$ para cada símbolo $a \in \Sigma$.

Se puede demostrar entonces que $L(M) = L(G)$, es decir, M y G son equivalentes. Diremos entonces que M es el autómata con pila asociado a G.

Algoritmo para construir un autómata con pila equivalente a una gramática incontextual.

Si $G = (V, \Sigma, P, S)$ es una gramática incontextual y $M = (\{q_0, f\}, \Sigma, V \cup \Sigma, \Delta, q_0, \{f\})$ es el autómata con pila asociado a G según la anterior definición y queremos reconocer una palabra $x \in \Sigma^*$ en el autómata M, procederemos de la siguiente manera:

Aplicamos la transición de tipo 1 de M, con lo cual pasamos al estado aceptador f y ponemos el símbolo inicial S en la pila. A continuación, cuando aparezca un símbolo t en la cinta de entrada, aplicaremos transiciones de tipo 2 del autómata M hasta generar el símbolo t en la cima de la pila. Entonces, aplicamos la transición de tipo 3 $((f, t, t), (f, \lambda))$, con lo cual leemos el símbolo t en la cinta de entrada, y cancelamos ese símbolo en la pila. La idea, por tanto, es que para leer un símbolo t en la cinta de entrada, hay que generarlo en la cima de la pila para poder aplicar una transición de tipo 3.

Ejemplo 1

Consideremos la gramática incontextual $G = (V, \Sigma, P, S)$ donde $V = \{S\}$, $\Sigma = \{a, b, c\}$ y $P = \{S \longrightarrow aSa, S \longrightarrow bSb, S \longrightarrow c\}$.

El autómata con pila M asociado a G es entonces $M = (\{q_0, f\}, \{a, b, c\}, \{S, a, b, c\}, \Delta, q_0, \{f\})$ donde Δ consta de las siguientes transiciones:

1. $((q_0, \lambda, \lambda), (f, S))$.
2. $((f, \lambda, S), (f, aSa))$.
3. $((f, \lambda, S), (f, bSb))$.
4. $((f, \lambda, S), (f, c))$.
5. $((f, a, a), (f, \lambda))$.
6. $((f, b, b), (f, \lambda))$.
7. $((f, c, c), (f, \lambda))$.

Veamos ahora el cómputo en el autómata para la entrada $x = abbcbba$.

Ejemplo 1

estado	cinta	pila	transición
q_0	abbcbba	λ	–
f	abbcbba	S	1
f	abbcbba	aSa	2
f	bbcbba	Sa	5
f	bbcbba	bSba	3
f	bcbba	Sba	6
f	bcbba	bSbba	3
f	cbba	Sbba	6
f	cbba	cbba	4
f	bba	bba	7
f	ba	ba	6
f	a	a	6
f	λ	λ	5

Ejemplo 2

Consideremos la gramática incontextual G dada por las siguientes producciones:

1. $S \longrightarrow (S)$
2. $S \longrightarrow SS$
3. $S \longrightarrow \lambda$

Se tiene que $L(G)$ es el lenguaje de las palabras de paréntesis balanceados, es decir, el lenguaje de las palabras $x \in \{(,)\}^*$ tales que a todo paréntesis abierto en x le corresponde un paréntesis cerrado.

Esta gramática fue estudiada en la clase anterior.

El autómata con pila M asociado a G es entonces $M = (\{q_0, f\}, \{(,)\}, \{S, (,)\}, \Delta, q_0, \{f\})$ donde Δ consta de las siguientes transiciones:

Ejemplo 2

1. $((q_0, \lambda, \lambda), (f, S))$.
2. $((f, \lambda, S), (f, (S)))$.
3. $((f, \lambda, S), (f, SS))$.
4. $((f, \lambda, S), (f, \lambda))$.
5. $((f, (, ()), (f, \lambda))$.
6. $((f,),)), (f, \lambda))$.

Veamos ahora el cómputo en el autómata para la entrada $x = (()())$.

Ejemplo 2

estado	cinta	pila	transición
q_0	$(()())$	λ	–
f	$(()())$	S	1
f	$(()())$	(S)	2
f	$()())$	$S)$	5
f	$()())$	$SS)$	3
f	$()())$	$(S)S)$	2
f	$)())$	$S)S)$	5
f	$)())$	$)S)$	4
f	$())$	$S)$	6
f	$())$	$(S))$	2
f	$))$	$S))$	5
f	$))$	$))$	4
f	$)$	$)$	6
f	λ	λ	6

Gramáticas ambiguas

Sea $G = (V, \Sigma, P, S)$ una gramática incontextual. A toda derivación $S \Rightarrow_G^* x$ en G le asociamos el siguiente árbol de derivación:

(1) La raíz del árbol es S.

(2) Cada nodo del árbol que no sea una hoja corresponde a una variable de la gramática que es sustituida en el proceso de derivación.

(3) El producto del árbol, es decir la palabra formada por las hojas en sentido de izquierda a derecha, es x.

A dicho árbol se le llama árbol de derivación de x.

Gramáticas ambiguas

Decimos entonces que la gramática G es ambigua, si hay una palabra $x \in L(G)$ que tiene más de un árbol de derivación.

Las gramáticas ambiguas no se deben utilizar en el diseño de compiladores, porque si se utilizaran habría instrucciones en los lenguajes de programación que se podrían interpretar de diferentes maneras y dar resultados de ejecución diferentes.

Ejemplo 1

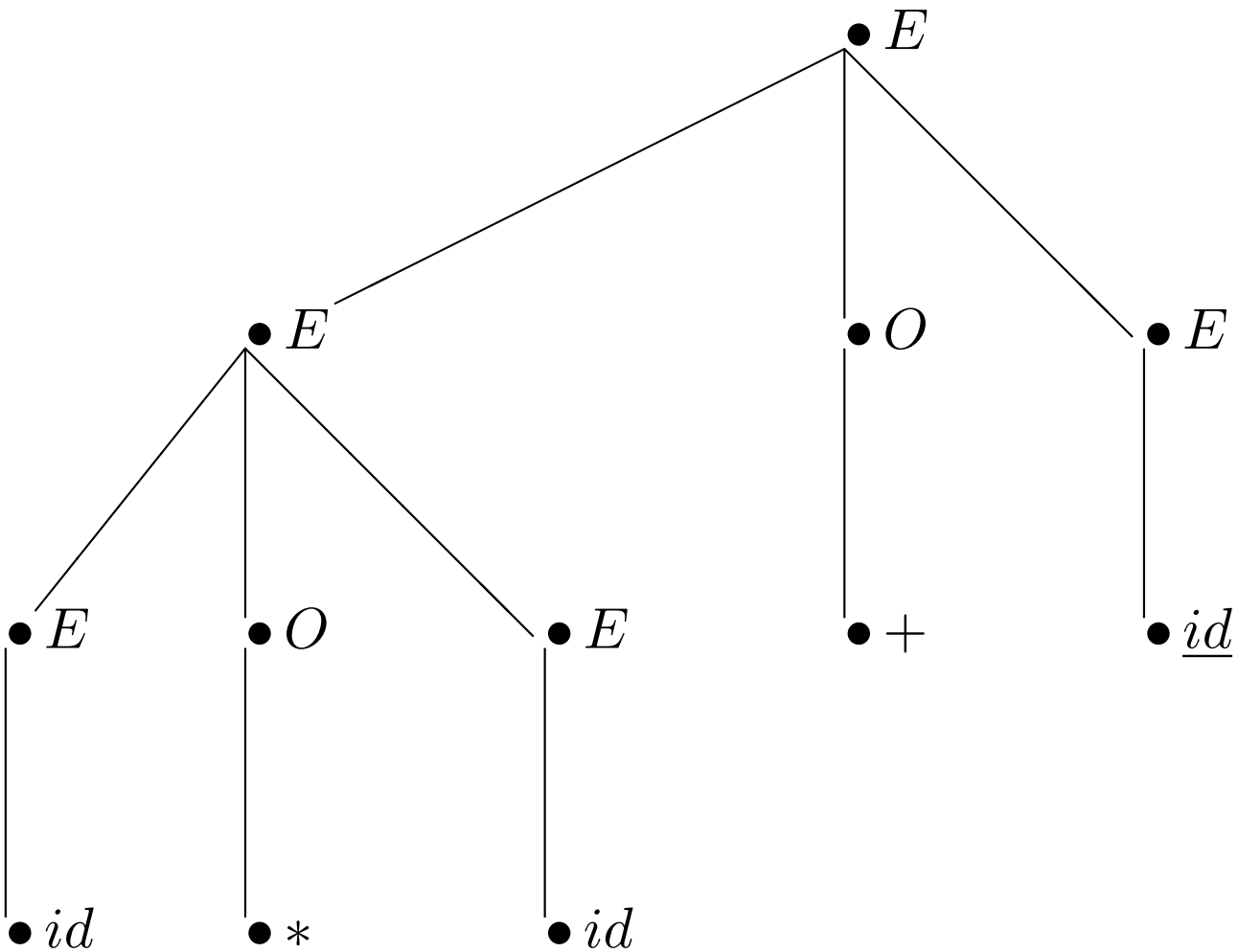
Consideremos la siguiente gramática incontextual G para generar expresiones aritméticas en donde aparezcan variables y las operaciones $+, -, *$ y $/$. Como estamos en la fase de análisis sintáctico del compilador, representamos a las variables por la categoría sintáctica "identificador", que denotamos por $\underline{\text{id}}$. Definimos entonces la gramática $G = (V, \Sigma, P, S)$ donde $V = \{E, O\}$, $\Sigma = \{\underline{id}, +, -, *, /\}$, $S = E$ y P consta de las siguientes producciones:

1. $E \longrightarrow E\,O\,E$
2. $O \longrightarrow +$
3. $O \longrightarrow -$
4. $O \longrightarrow *$
5. $O \longrightarrow /$
6. $E \longrightarrow \underline{id}$
7. $E \longrightarrow (E)$

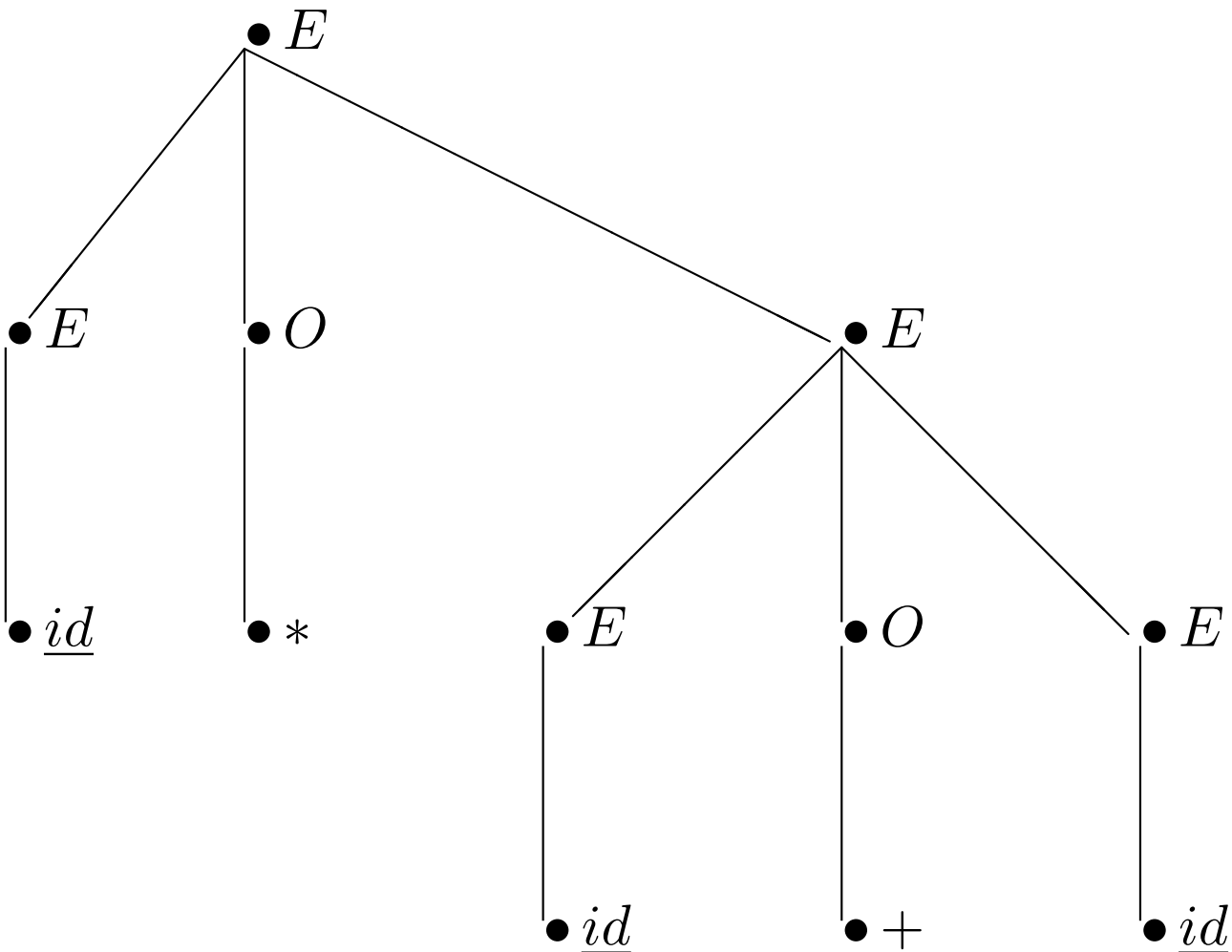
Ejemplo 1

Consideremos la palabra $x = \underline{id} * \underline{id} + \underline{id}$. Tenemos entonces los dos siguientes árboles de derivación para x.

Ejemplo 1



Ejemplo 1



Ejemplo 1

Por tanto, la palabra $x = \underline{id} * \underline{id} + \underline{id}$ tiene dos representaciones distintas en la gramática G, que corresponden a los dos árboles de derivación.

Si tomamos la primera representación, el significado de x sería $(\underline{id} * \underline{id}) + \underline{id}$, que es el siginificado correcto.

Y si tomamos la segunda representación, el significado de x sería $\underline{id} * (\underline{id} + \underline{id})$, que es incorrecto.

Por tanto, la gramática G es ambigua.

Sin embargo, la siguiente gramática G' para generar expresiones aritméticas donde aparecen variables no es ambigua:

Ejemplo 1

1. $E \rightarrow E + T$
2. $E \rightarrow E - T$
3. $E \rightarrow T$
4. $T \rightarrow T * F$
5. $T \rightarrow T \,/\, F$
6. $T \rightarrow F$
7. $F \rightarrow (E)$
8. $F \rightarrow \underline{id}$

Se puede comprobar fácilmente que la gramática G' no es ambigua, ya que con la variable E se genera una suma/resta de términos de manera unívoca, con la variable T se genera un producto/división de factores también de manera unívoca, y con la variable F se generan los átomos de las expresiones aritméticas, que pueden ser expresiones aritméticas entre paréntesis o identificadores.

Ejemplo 2

Demostramos que es ambigua la gramática incontextual definida por las dos siguientes producciones, para generar las instrucciones condicionales de un lenguaje de programación.

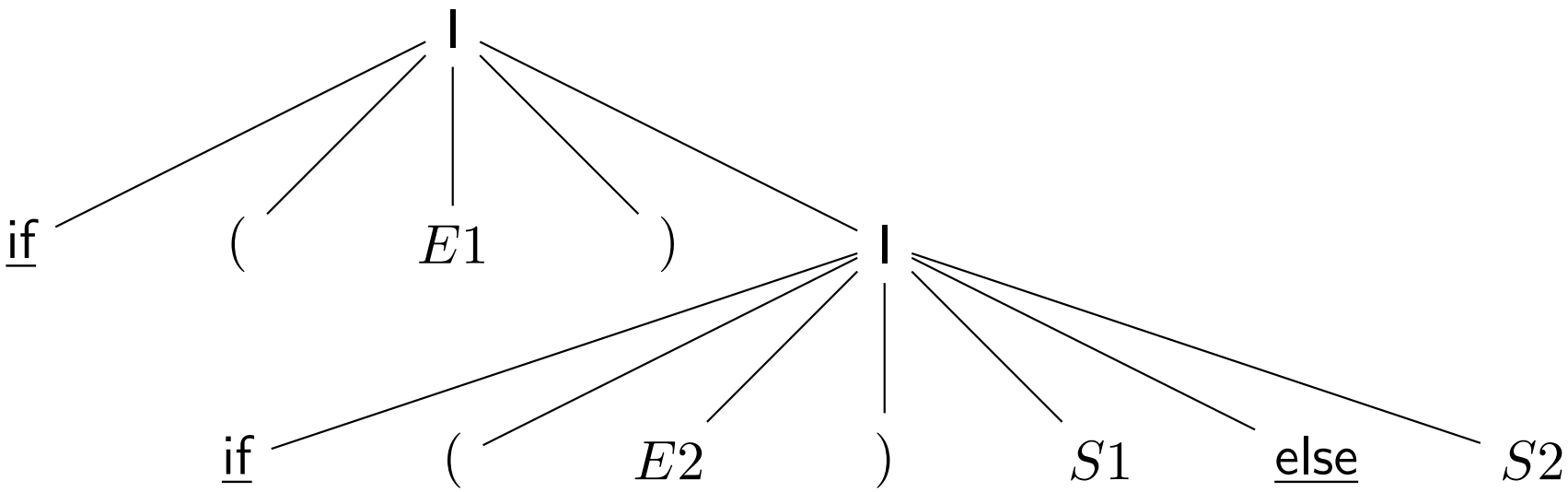
1. $I \longrightarrow \underline{if}\,(E)\,I\,\underline{else}\,I$
2. $I \longrightarrow \underline{if}\,(E)\,I$

La gramática es ambigua, porque una instrucción de la forma

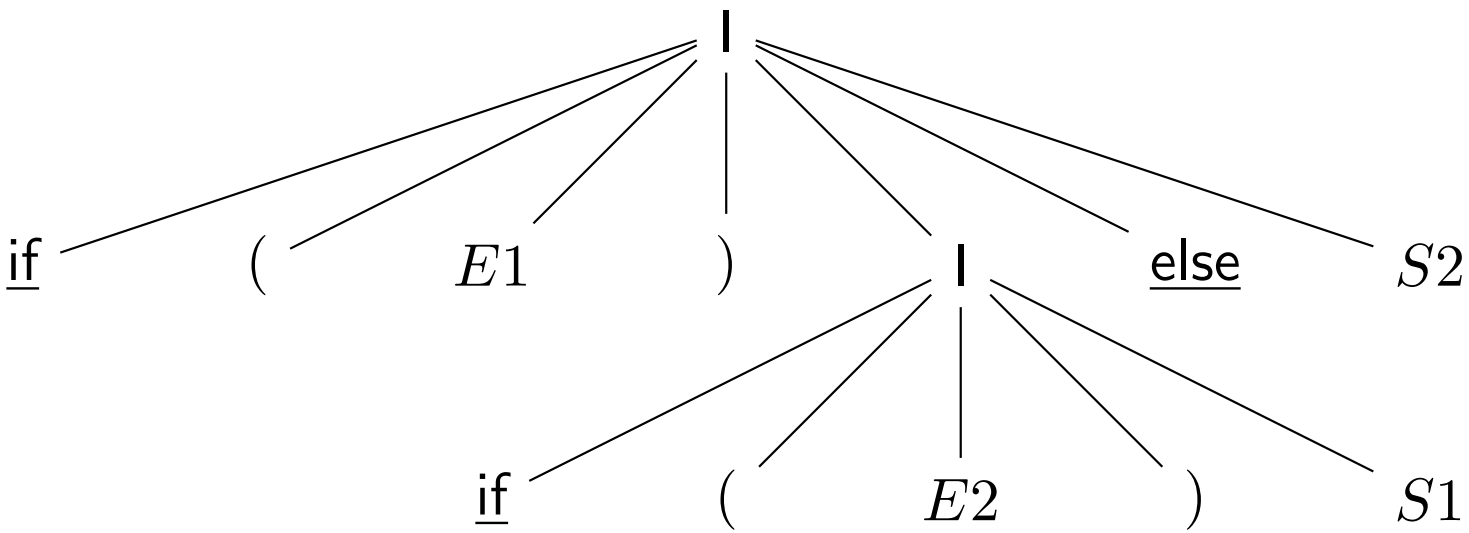
$$\underline{if}\,(E1)\,\underline{if}\,(E2)\,S1\,\underline{else}\,S2$$

(es decir, una instrucción con dos ifs y un solo else) tiene los dos siguientes árboles de derivación:

Ejemplo 2



Ejemplo 2



Ejemplo 2

Se observa que en el primer árbol de derivación, el else va asociado con el if interno. Mientras que en el segundo árbol de derivación, el else va asociado con el if externo.

Por tanto, el segundo árbol de derivación es incorrecto, ya que en una instrucción condicional con dos if y un else, el else va siempre asociado al if interno.

Por consiguiente, sólo el primer árbol de derivación es correcto.

Ejemplo 2

Para obtener una gramática no ambigua que trate correctamente las instrucciones condicionales, hay que utilizar dos variables, en lugar de una única variable I. Por ejemplo, se puede demostrar que la siguiente gramática que genera instrucciones condicionales no es ambigua:

1. $I \longrightarrow \underline{if}\,(E)\,J\,\underline{else}\,I$
2. $I \longrightarrow \underline{if}\,(E)\,I$
3. $J \longrightarrow \underline{if}\,(E)\,J\,\underline{else}\,J$

En esta última gramática, observamos que hay una variable I que genera instrucciones condicionales simples aplicando la producción 2, y hay una variable J que genera instrucciones condicionales compuestas aplicando la producción 3. Mientras que en la anterior gramática hay una única variable I, que genera tanto instrucciones condicionales simples como compuestas, y de ahí proviene la ambigüedad de la gramática.

Algoritmo para diseñar un analizador sintáctico

Consta de las siguientes tres fases:

(1) Diseñar una gramática incontextual no ambigua que genere el lenguaje que estemos considerando.

(2) Transformar la gramática del paso (1) en una gramática equivalente cuyo autómata con pila asociado sea determinista.

(3) Programar el autómata con pila asociado a la gramática del paso (2).

El programa del paso (3) será entonces el analizador sintáctico.

La fase (2) es la fase más compleja del algoritmo, y de ella nos ocuparemos a partir de ahora.

Gramáticas LL(1)

Para poder diseñar un analizador sintáctico es necesario partir de una gramática no ambigua, ya que si no lo hiciéramos, habría instrucciones del lenguaje de programación que se podrían interpretar de diferentes maneras y dar resultados de ejecución diferentes.

Nos interesa entonces tener gramáticas no ambiguas cuyos autómatas con pila asociados sean programables. Una clase importante de gramáticas no ambiguas con esta condición es la clase de las llamadas gramáticas LL(1).

Aunque dichas gramáticas no son las únicas que se utilizan en el diseño de compiladores, es posible diseñar una parte importante del analizador sintáctico del compilador utilizando únicamente gramáticas LL(1).

Para poder definir el concepto de gramática LL(1), necesitamos definir previamente el concepto de tabla de análisis de una gramática.

Las funciones Primeros y Siguientes

Sea $G = (V, \Sigma, P, S)$ una gramática incontextual.

(a) Para toda palabra $\alpha \in (V \cup \Sigma)^*$ definimos el conjunto Primeros(α) de la siguiente manera. Si $\alpha \not\Rightarrow_G^* \lambda$, definimos

$$\text{Primeros}(\alpha) = \{a \in \Sigma : \text{ existe una palabra } \beta \text{ tal que } \alpha \Rightarrow_G^* a\beta\}.$$

Y si $\alpha \Rightarrow_G^* \lambda$, definimos

$$\text{Primeros}(\alpha) = \{a \in \Sigma : \text{ existe una palabra } \beta \text{ tal que } \alpha \Rightarrow_G^* a\beta\} \cup \{\lambda\}.$$

(b) Para toda variable $X \in V$, definimos

$$\text{Siguientes}(X) = \{a \in \Sigma : \text{ existen palabras } \alpha, \beta \text{ tales que } S \Rightarrow_G^* \alpha X a \beta\}.$$

Definición de gramática LL(1)

(c) Si $X \in V$ y $a \in \Sigma$, definimos

TABLA$[X, a] = \{(X, \beta) \in P : a \in$ Primeros$(\beta \cdot$ Siguientes(X))$\}$.

(d) Decimos entonces que la gramática G es LL(1), si el algoritmo de construcción de la tabla de análisis no genera conflictos, es decir, si para todo $X \in V$ y para todo $a \in \Sigma$ se tiene que TABLA$[X, a]$ no contiene más de un elemento.

Se puede entonces demostrar que toda gramática LL(1) es una gramática no ambigua.

Observación

Supongamos que $X \longrightarrow \lambda$ es una regla de una gramática incontextual G, y queremos saber en qué casillas de la tabla de análisis está la regla $X \longrightarrow \lambda$. Obsérvese que en la fórmula para construir la tabla de análisis, tenemos que

$\text{Primeros}(\lambda \cdot \text{ Siguientes}(X)) = \text{Primeros}(\text{ Siguientes}(X)) = \text{Siguientes}(X)$,

ya que todo elemento de $\text{Siguientes}(X)$ es un símbolo terminal, y para todo símbolo terminal t se tiene que $\text{Primeros}(t) = \{t\}$.

Por tanto, para saber en qué casillas de la tabla de análisis está la regla $X \longrightarrow \lambda$, tendremos que calcular los $\text{Siguientes}(X)$. Entonces, para todo símbolo $t \in \text{Siguientes}(X)$, tendremos que $X \longrightarrow \lambda \in \text{TABLA}(X, t)$.

Ejemplo

Consideremos la gramática incontextual G dada por las siguientes producciones:

1. $S \longrightarrow (S)$
2. $S \longrightarrow SS$
3. $S \longrightarrow \lambda$

Esta gramática G la estudiamos en la última clase

Se tiene que $L(G)$ es el lenguaje de las palabras de paréntesis balanceados, es decir, el lenguaje de las palabras $x \in \{(,)\}^*$ tales que a todo paréntesis abierto en x le corresponde un paréntesis cerrado.

La tabla de análisis para la gramática G, que se construye aplicando la fórmula vista anteriormente, es entonces la siguiente:

Ejemplo

TABLA	(	)
S	1,2	3

Obsérvese que de la derivación

$SS \Rightarrow^1 (S)S$

deducimos que $($ ∈ Primeros(SS).

Asimismo, de la derivación

$S \Rightarrow^2 SS \Rightarrow^1 (S)S$

deducimos que $)$ ∈ Siguientes(S).

Ejemplo

Se observa entonces que la producción $1 \in$ TABLA$[S,(]$, porque $(\in$ Primeros$((S))$.

La producción $2 \in$ TABLA$[S,(]$, porque $(\in$ Primeros(SS).

Y la producción $3 \in$ TABLA$[S,)]$, porque $) \in$ Siguientes(S).

Como las producciones 1 y 2 pertenecen a TABLA$[S,(]$, hay un conflicto en la tabla de análisis de G, por lo cual G no es LL(1).

En este caso, se puede eliminar el conflicto directamente, sustituyendo las producciones 1 y 2 de G por la producción $S \longrightarrow (S)S$. Obtenemos entonces la gramática G' equivalente a G dada por las dos siguientes producciones:

Ejemplo

1. $S \longrightarrow (S)S$
2. $S \longrightarrow \lambda$

La tabla de análisis para la gramática G', que se construye aplicando la fórmula vista anteriormente, es entonces la siguiente:

Ejemplo

TABLA	(	)
S	1	2

Se observa entonces que la producción $1 \in$ TABLA$[S,($], porque $(\in$ Primeros$((S)S)$.

Y la producción $2 \in$ TABLA$[S,)]$, porque $) \in$ Siguientes(S).

Por tanto, la gramática G' es LL(1), ya que no hay conflictos en su tabla de análisis.

Eliminación del indeterminismo en gramáticas LL(1)

Las gramáticas LL(1) son interesantes para el diseño de compiladores, porque utilizando la tabla de análisis de la gramática se puede eliminar fácilmente el indeterminismo del autómata con pila asociado a la gramática.

Para ello, si en un paso de cómputo del autómata el tope de la pila es una variable X, el carácter de entrada es a y hay varias transiciones del autómata con pila que pueden aplicarse, consultamos Tabla$[X, a]$. Si Tabla$[X, a]$ no contiene ninguna producción, se devuelve "error". Si no, se aplica la transición del autómata correspondiente a la producción de Tabla$[X, a]$.

Los autómatas con pila asociados a gramáticas LL(1) se pueden entonces programar mediante el siguiente algoritmo.

Programación de autómatas con pila asociados a gramáticas LL(1)

public boolean analisis_sintactico (String entrada)
{ Stack⟨Character⟩ pila = new Stack⟨Character⟩();
int q = 0, i = 0; boolean b = true; char c = entrada.charAt(0);
(1) Poner '$' en la pila
(2) Aplicar la transicion $((q_0, \lambda, \lambda), (f, S))$
(3) while ((tope de la pila ! = '$' || caracter de entrada ! = '$')
&& b)
{ if (tope de la pila == 'a' && caracter de entrada == 'a')
{aplicar la transicion $((f, a, a), (f, \lambda))$; leer siguiente caracter;}
else if (tope de la pila == 'X' && caracter de entrada == 'a'
&&TABLA$[X, a] == X \longrightarrow X1 \dots X_n$)
aplicar la transicion $((f, \lambda, X), (f, X1 \dots X_n))$;
else b = false; }
(4) if (tope de la pila == '$' && caracter de entrada == '$')
return true; else return false;
}