LENGUAJES INCONTEXTUALES

Abril y Mayo de 2023

Introducción

Utilizando los autómatas que estudiamos en el tema anterior, los autómatas finitos, vimos cómo se puede diseñar el analizador léxico de un compilador. Sin embargo, los autómatas finitos no se pueden utilizar para diseñar el analizador sintáctico del compilador. Para ello, nos hace falta utilizar autómatas más complejos, que son los llamados autómatas con pila, los cuales tienen asociada una pila que utilizan como elemento de memoria. Al igual que los autómatas finitos, los autómatas con pila tienen también asociada una cinta de entrada dividida en celdas, en las cuales se registra la palabra de entrada que suministremos al autómata.

Definición de un autómata con pila

Un autómata con pila es una estructura $M = (K, \Sigma, \Gamma, \Delta, q_0, F)$ donde:

(1) K es un conjunto finito y no vacío de estados.

(2) Σ es un alfabeto finito, el alfabeto de entrada.

(3) Γ es un alfabeto finito, el alfabeto de la pila.

(4) $q_0 \in K$ es el estado inicial,

(5) $F \subseteq K$ es el conjunto de estados aceptadores (o estados finales),

(6) Δ es un conjunto finito de elementos de la forma $((p, a, b), (q, x))$ donde $p, q \in K$, $a \in \Sigma \cup \{\lambda\}$, $b \in \Gamma \cup \{\lambda\}$ y $x \in \Gamma^*$.

Conceptos básicos

Si $M = (K, \Sigma, \Gamma, \Delta, q_0, F)$ es un autómata con pila, llamaremos transiciones a los elementos de Δ.

Al empezar el cómputo en un autómata con pila $M = (K, \Sigma, \Gamma, \Delta, q_0, F)$, estaremos en el estado inicial q_0, tendremos una palabra $x \in \Sigma^*$ en la cinta de entrada y tendremos la pila vacía.

Si en un paso de cómputo de M, queremos aplicar una transición $((p, a, b), (q, x))$ donde $a \in \Sigma$ y $b \in \Gamma$, el símbolo b tiene que estar situado en la cima de la pila. Al aplicar entonces la transición, sucede lo siguiente:

(a) se pasa del estado p al estado q,

(b) se lee el símbolo a de la cinta de entrada,

(c) se reemplaza en la pila el símbolo b por x.

Conceptos básicos

Análogamente, si en un paso de cómputo de M, queremos aplicar una transición $((p, \lambda, b), (q, x))$ donde $b \in \Gamma$, el símbolo b tiene que estar situado en la cima de la pila. Al aplicar entonces la transición, sucede lo siguiente:

(a) se pasa del estado p al estado q,

(b) no se lee ningún símbolo en la cinta de entrada,

(c) se reemplaza en la pila el símbolo b por x.

De manera similar se procede con los demás tipos de transiciones.

Conceptos básicos

Si $M = (K, \Sigma, \Gamma, \Delta, q_0, F)$ es un autómata con pila, definimos una configuración de M como una palabra $\alpha p x \in \Gamma^* K \Sigma^*$.

Si en un paso de cómputo la configuración de M es $\alpha p x$, esto significa que el autómata M se encuentra en el estado p, x es la parte de la palabra de entrada que todavía no se ha leído y α es el contenido de la pila.

La noción de configuración nos permite entonces definir la noción de cómputo en un autómata con pila.

Cómputos en autómatas con pila

Supongamos que $M = (K, \Sigma, \Gamma, \Delta, q_0, F)$ es un autómata con pila y que $\alpha p x, \beta q y$ son configuraciones de M. Entonces:

(1) Decimos que $\alpha p x$ produce $\beta q y$ en un paso de cómputo, lo que representamos por $\alpha p x \vdash_M \beta q y$, si en M podemos pasar de $\alpha p x$ a $\beta q y$ aplicando una transición de Δ.

(2) Decimos que $\alpha p x$ produce $\beta q y$, lo que representamos por $\alpha p x \vdash_M^* \beta q y$, si en M podemos pasar de $\alpha p x$ a $\beta q y$ aplicando un número finito de de transiciones de Δ.

Lenguaje asociado a un autómata con pila

Supongamos que $M = (K, \Sigma, \Gamma, \Delta, q_0, F)$ es un autómata con pila.

(1) Una palabra $x \in \Sigma^*$ es reconocida (o aceptada) por M, si existe $q \in F$ tal que $\lambda q_0 x \vdash_M^* \lambda q \lambda$.

(2) Definimos el lenguaje reconocido (o aceptado) por M por

$$L(M) = \{x \in \Sigma^* : x \text{ es reconocida por } M\}.$$

Por tanto, para que una palabra sea reconocida por un autómata con pila M, tiene que haber un cómputo en el autómata M que lea toda la palabra de entrada de manera que al final del cómputo se llegue a un estado aceptador y la pila quede vacía.

Ejemplo

Consideremos el autómata con pila $M = (K, \Sigma, \Gamma, \Delta, q_0, F)$ donde $K = \{q_0, f\}$, $\Sigma = \{a, b, c\}$, $\Gamma = \{a, b\}$, $F = \{f\}$ y Δ consta de las siguientes transiciones:

1. $((q_0, a, \lambda), (q_0, a))$.
2. $((q_0, b, \lambda), (q_0, b))$.
3. $((q_0, c, \lambda), (f, \lambda))$.
4. $((f, a, a), (f, \lambda))$.
5. $((f, b, b), (f, \lambda))$.

Tenemos entonces el siguiente cómputo para la entrada $abbcbba$.

Ejemplo

estado	cinta	pila	transición
q_0	abbcbba	λ	–
q_0	bbcbba	a	1
q_0	bcbba	ba	2
q_0	cbba	bba	2
f	bba	bba	3
f	ba	ba	5
f	a	a	5
f	λ	λ	4

Ejemplo

Se observa que el autómata M, cuando está en el estado q_0, mete una a en la pila cada vez que lee una a en la cinta aplicando la transición 1, y mete una b en la pila cada vez que lee una b en la cinta aplicando la transición 2. Y así actúa el autómata hasta que entra una c en la cinta, en cuyo caso pasa al estado aceptador f aplicando la transición 3. Una vez que el autómata está en el estado f, cancela las a's que entren en la cinta con las a's que se encuentran en la pila aplicando la transición 4, y asimismo cancela las b's que entren en la cinta con las b's que se encuentran en la pila aplicando la transición 5. Entonces, para que al final del cómputo la pila quede vacía, ha de suceder que la parte de la entrada que aparezca después de la c en la entrada ha de coincidir con la inversa de la parte de la entrada que aparece antes de la c en la entrada. Por tanto,

$$L(M) = \{xcx^I : x \in \{a, b\}^*\}.$$

Introducción

Utilizando los autómatas que estudiamos en el tema anterior, los autómatas finitos, vimos cómo se puede diseñar el analizador léxico de un compilador. Sin embargo, los autómatas finitos no se pueden utilizar para diseñar el analizador sintáctico del compilador. Para ello, nos hace falta utilizar autómatas más complejos, que son los llamados autómatas con pila, los cuales tienen asociada una pila que utilizan como elemento de memoria. Al igual que los autómatas finitos, los autómatas con pila tienen también asociada una cinta de entrada dividida en celdas, en las cuales se registra la palabra de entrada que suministremos al autómata.

Al final de la última clase, empezamos el estudio de los autómatas con pila. En la clase de hoy, estudiaremos con más profundidad estos autómatas. Empezamos recordando el concepto de autómata con pila.

Definición de un autómata con pila

Un autómata con pila es una estructura $M = (K, \Sigma, \Gamma, \Delta, q_0, F)$ donde:

(1) K es un conjunto finito y no vacío de estados.

(2) Σ es un alfabeto finito, el alfabeto de entrada.

(3) Γ es un alfabeto finito, el alfabeto de la pila.

(4) $q_0 \in K$ es el estado inicial,

(5) $F \subseteq K$ es el conjunto de estados aceptadores (o estados finales),

(6) Δ es un conjunto finito de elementos de la forma $((p, a, b), (q, x))$ donde $p, q \in K$, $a \in \Sigma \cup \{\lambda\}$, $b \in \Gamma \cup \{\lambda\}$ y $x \in \Gamma^*$.

Conceptos básicos

Si $M = (K, \Sigma, \Gamma, \Delta, q_0, F)$ es un autómata con pila, llamaremos transiciones a los elementos de Δ.

Al empezar el cómputo en un autómata con pila $M = (K, \Sigma, \Gamma, \Delta, q_0, F)$, estaremos en el estado inicial q_0, tendremos una palabra $x \in \Sigma^*$ en la cinta de entrada y tendremos la pila vacía.

Si en un paso de cómputo de M, queremos aplicar una transición $((p, a, b), (q, x))$ donde $a \in \Sigma$ y $b \in \Gamma$, el símbolo b tiene que estar situado en la cima de la pila. Al aplicar entonces la transición, sucede lo siguiente:

(a) se pasa del estado p al estado q,

(b) se lee el símbolo a de la cinta de entrada,

(c) se reemplaza en la pila el símbolo b por x.

Conceptos básicos

Análogamente, si en un paso de cómputo de M, queremos aplicar una transición $((p, \lambda, b), (q, x))$ donde $b \in \Gamma$, el símbolo b tiene que estar situado en la cima de la pila. Al aplicar entonces la transición, sucede lo siguiente:

(a) se pasa del estado p al estado q,

(b) no se lee ningún símbolo en la cinta de entrada,

(c) se reemplaza en la pila el símbolo b por x.

De manera similar se procede con los demás tipos de transiciones.

Conceptos básicos

Si $M = (K, \Sigma, \Gamma, \Delta, q_0, F)$ es un autómata con pila, definimos una configuración de M como una palabra $\alpha p x \in \Gamma^* K \Sigma^*$.

Si en un paso de cómputo la configuración de M es $\alpha p x$, esto significa que el autómata M se encuentra en el estado p, x es la parte de la palabra de entrada que todavía no se ha leído y α es el contenido de la pila.

La noción de configuración nos permite entonces definir la noción de cómputo en un autómata con pila.

Cómputos en autómatas con pila

Supongamos que $M = (K, \Sigma, \Gamma, \Delta, q_0, F)$ es un autómata con pila y que $\alpha px, \beta qy$ son configuraciones de M. Entonces:

(1) Decimos que αpx produce βqy en un paso de cómputo, lo que representamos por $\alpha px \vdash_M \beta qy$, si en M podemos pasar de αpx a βqy aplicando una transición de Δ.

(2) Decimos que αpx produce βqy, lo que representamos por $\alpha px \vdash_M^* \beta qy$, si en M podemos pasar de αpx a βqy aplicando un número finito de de transiciones de Δ.

Lenguaje asociado a un autómata con pila

Supongamos que $M = (K, \Sigma, \Gamma, \Delta, q_0, F)$ es un autómata con pila.

(1) Una palabra $x \in \Sigma^*$ es reconocida (o aceptada) por M, si existe $q \in F$ tal que $\lambda q_0 x \vdash_M^* \lambda q \lambda$.

(2) Definimos el lenguaje reconocido (o aceptado) por M por

$$L(M) = \{x \in \Sigma^* : x \text{ es reconocida por } M\}.$$

Por tanto, para que una palabra sea reconocida por un autómata con pila M, tiene que haber un cómputo en el autómata M que lea toda la palabra de entrada de manera que al final del cómputo se llegue a un estado aceptador y la pila quede vacía.

Ejemplo

Consideremos el autómata con pila $M = (K, \Sigma, \Gamma, \Delta, q_0, F)$ donde $K = \{q_0, f\}$, $\Sigma = \{a, b, c\}$, $\Gamma = \{a, b\}$, $F = \{f\}$ y Δ consta de las siguientes transiciones:

1. $((q_0, a, \lambda), (q_0, a))$.
2. $((q_0, b, \lambda), (q_0, b))$.
3. $((q_0, c, \lambda), (f, \lambda))$.
4. $((f, a, a), (f, \lambda))$.
5. $((f, b, b), (f, \lambda))$.

Tenemos entonces el siguiente cómputo para la entrada $abbcbba$.

Ejemplo

estado	cinta	pila	transición
q_0	abbcbba	λ	–
q_0	bbcbba	a	1
q_0	bcbba	ba	2
q_0	cbba	bba	2
f	bba	bba	3
f	ba	ba	5
f	a	a	5
f	λ	λ	4

Ejemplo

Se observa que el autómata M, cuando está en el estado q_0, mete una a en la pila cada vez que lee una a en la cinta aplicando la transición 1, y mete una b en la pila cada vez que lee una b en la cinta aplicando la transición 2. Y así actúa el autómata hasta que entra una c en la cinta, en cuyo caso pasa al estado aceptador f aplicando la transición 3. Una vez que el autómata está en el estado f, cancela las a's que entren en la cinta con las a's que se encuentran en la pila aplicando la transición 4, y asimismo cancela las b's que entren en la cinta con las b's que se encuentran en la pila aplicando la transición 5. Entonces, para que al final del cómputo la pila quede vacía, ha de suceder que la parte de la entrada que aparezca después de la c en la entrada ha de coincidir con la inversa de la parte de la entrada que aparece antes de la c en la entrada. Por tanto,

$$L(M) = \{xcx^I : x \in \{a, b\}^*\}.$$

Programación de autómatas con pila deterministas

Obsérvese que el que dos transiciones sean compatibles en un autómata con pila significa que las dos se pueden aplicar en un mismo paso de cómputo del autómata. Por tanto, en cualquier paso de cómputo de un autómata con pila determinista habrá a lo sumo una transición que se pueda aplicar. Esto hace que los autómatas con pila deterministas se puedan programar. Para ello, se procede de manera similar a como vimos con los autómatas deterministas finitos. Si programamos un autómata con pila determinista en Java, utilizaremos una variable tipo stack de Java para manejar la pila asociada al autómata y utilizaremos una variable booleana que tendrá el valor verdadero cuando el cómputo del autómata quede bloqueado (por no poder aplicar ninguna transición del autómata), en cuyo caso se saldrá del bucle "while" del programa para devolver el valor "false".

Autómatas con pila deterministas

Nuestro objetivo ahora es definir el concepto de autómata con pila determinista, que es el modelo de autómata con pila que se puede programar directamente. Previamente, necesitamos definir cuando dos transiciones son compatibles.
Sea $M = (K, \Sigma, \Gamma, \Delta, q_0, F)$ un autómata con pila.

(a) Dos transiciones $((p_1, a_1, b_1), (q_1, \alpha_1))$ y $((p_2, a_2, b_2), (q_2, \alpha_2))$ de M son compatibles, si se cumplen las tres siguientes condiciones:

(1) $p_1 = p_2$.

(2) $a_1 = a_2$ o $a_1 = \lambda$ o $a_2 = \lambda$.

(3) $b_1 = b_2$ o $b_1 = \lambda$ o $b_2 = \lambda$.

(b) M es determinista, si no tiene dos transiciones compatibles distintas.

Ejemplo 1

Consideremos el autómata con pila M visto en el ejemplo anterior. Es decir, $M = (K, \Sigma, \Gamma, \delta, q_0, F)$ donde $K = \{q_0, f\}$, $\Sigma = \{a, b, c\}$, $\Gamma = \{a, b\}$, $F = \{f\}$ y Δ consta de las siguientes transiciones:

1. $((q_0, a, \lambda), (q_0, a))$.
2. $((q_0, b, \lambda), (q_0, b))$.
3. $((q_0, c, \lambda), (f, \lambda))$.
4. $((f, a, a), (f, \lambda))$.
5. $((f, b, b), (f, \lambda))$.

Se tiene entonces que M es determinista, porque no tiene dos transiciones compatibles distintas.

Ejemplo 2

Consideremos ahora el autómata con pila $M' = (K, \Sigma, \Gamma, \delta, q_0, F)$ donde $K = \{q_0, f\}$, $\Sigma = \{a, b\}$, $\Gamma = \{a, b\}$, $F = \{f\}$ y Δ consta de las siguientes transiciones:

1. $((q_0, a, \lambda), (q_0, a))$.
2. $((q_0, b, \lambda), (q_0, b))$.
3. $((q_0, \lambda, \lambda), (f, \lambda))$.
4. $((f, a, a), (f, \lambda))$.
5. $((f, b, b), (f, \lambda))$.

Tenemos entonces que M' no es determinista, ya que las transiciones 1 y 3 son compatibles, y asimismo las transiciones 2 y 3 son compatibles.

Los autómatas M y M' son muy similares. Su única diferencia está en la transición 3, que en M es $((q_0, c, \lambda), (f, \lambda))$, y en M' es $((q_0, \lambda, \lambda), (f, \lambda))$. El resto de las transiciones son las mismas en M y en M'.

Ejemplo 2

Por tanto,

$$L(M') = \{xx^I : x \in \{a, b\}^*\}.$$

El siguiente cómputo en M' reconoce la palabra $abbbba$.

Ejemplo 2

estado	cinta	pila	transición
q_0	abbbba	λ	–
q_0	bbbba	a	1
q_0	bbba	ba	2
q_0	bba	bba	2
f	bba	bba	3
f	ba	ba	5
f	a	a	5
f	λ	λ	4

Gramáticas incontextuales

Como ya hemos indicado anteriormente, los autómatas con pila se utilizan para poder diseñar el analizador sintáctico del compilador. Sin embargo, los autómatas con pila son difíciles de diseñar directamente. Para poder diseñarlos, se utilizan las gramáticas incontextuales, que son estructuras equivalentes a los autómatas con pila, pero mucho más fáciles de diseñar.

Como veremos, toda gramática incontextual tiene asociado un autómata con pila equivalente a la gramática. Entonces, una vez que tengamos construida la gramática incontextual adecuada para el lenguaje que estemos considerando, tomaremos el autómata con pila equivalente a la gramática, y a continuación procederemos a eliminar el indeterminismo de dicho autómata.

Definición de gramática incontextual

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.

Derivaciones en gramáticas incontextuales

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

(1) Si $(A, x) \in P$, escribiremos $A \longrightarrow x$.

Una producción $A \longrightarrow x$ de G se aplica a una palabra $u \in (V \cup \Sigma)^*$ en la que aparece A. El efecto de aplicar la producción $A \longrightarrow x$ a u es sustituir en la palabra u una aparición de A por x. Por tanto, las producciones de una gramática se utilizan como reglas de reescritura.

(2) Para simplificar la notación, si $A \longrightarrow \alpha_1, \ldots, A \longrightarrow \alpha_n$ son reglas de una gramática incontextual con una misma parte izquierda, escribiremos $A \longrightarrow \alpha_1 | \ldots | \alpha_n$.

Derivaciones en gramáticas incontextuales

(3) Si $u, v \in (V \cup \Sigma)^*$, decimos que v se deriva de u en un paso, en símbolos $u \Rightarrow_G v$, si obtenemos v a partir de u aplicando una producción de P.

(4) Si $u, v \in (V \cup \Sigma)^*$, decimos que v se deriva de u, en símbolos $u \Rightarrow_G^* v$, si obtenemos v a partir de u aplicando un número finito de veces las producciones de P.

Lenguaje asociado a una gramática incontextual

Si $G = (V, \Sigma, P, S)$ es una gramática incontextual y $A \in V$, definimos $L(A) = \{x \in \Sigma^* : A \Rightarrow_G^* x\}$. Definimos entonces el lenguaje de G como

$$L(G) = L(S) = \{x \in \Sigma^* : S \Rightarrow_G^* x\}.$$

Decimos que un lenguaje L es incontextual, si existe una gramática incontextual G tal que $L(G) = L$.

Ejemplo 1

Consideremos la gramática incontextual $G = (V, \Sigma, P, S)$ donde:

(1) $V = \{S, X\}$,

(2) $\Sigma = \{0, 1, 2\}$ y

(3) P consta de las siguientes producciones:

1. $S \longrightarrow 0X$,
2. $S \longrightarrow 2$,
3. $X \longrightarrow 0S$,
4. $X \longrightarrow 1$.

Tenemos entonces las siguientes derivaciones en G:

Ejemplo 1

$S \Rightarrow^2 2$,

$S \Rightarrow^1 0X \Rightarrow^4 01$,

$S \Rightarrow^1 0X \Rightarrow^3 00S \Rightarrow^2 002$,

$S \Rightarrow^1 0X \Rightarrow^3 00S \Rightarrow^1 000X \Rightarrow^4 0001$,

$S \Rightarrow^1 0X \Rightarrow^3 00S \Rightarrow^1 000X \Rightarrow^3 0000S \Rightarrow^2 00002$,

$S \Rightarrow^1 0X \Rightarrow^3 00S \Rightarrow^1 000X \Rightarrow^3 0000S \Rightarrow^1 00000X \Rightarrow^4 000001$,

$\vdots$

$\vdots$

Por tanto, $L(G) = \{0^n 2 : n \text{ es par }\} \cup \{0^n 1 : n \text{ es impar }\}$.

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.

Por ejemplo, la palabra $x = (()())()$ es generada por la siguiente derivación:

$S \Rightarrow^2 SS \Rightarrow^1 S(S) \Rightarrow^3 S() \Rightarrow^1 (S)() \Rightarrow^2 (SS)() \Rightarrow^1 ((S)S)()$
$\Rightarrow^3 (()S)() \Rightarrow^1 (()(S))() \Rightarrow^3 (()())()$.

Gramáticas regulares

A continuación, estudiamos las llamadas gramáticas regulares, que tienen interés porque es sabido que mediante este tipo de gramáticas se pueden representar los tipos de datos de los lenguajes de programación.

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

Ejemplo

Consideremos la gramática regular G dada por las siguientes producciones:

1. $S \longrightarrow 1S$
2. $S \longrightarrow 0T$
3. $T \longrightarrow 0T$
4. $T \longrightarrow 1S$
5. $T \longrightarrow \lambda$

Tenemos entonces que si $S \Rightarrow^* xS$ donde $x \in \{0,1\}^*$, entonces la palabra x acaba en 1. Y si $S \Rightarrow^* xT$ donde $x \in \{0,1\}^*$, entonces la palabra x acaba en 0. Entonces, como T es la única variable que se anula (por la producción 5), deducimos que $L(G) = \{x \in \{0,1\}^* : x \text{ acaba en } 0\}$.

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 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.

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.

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

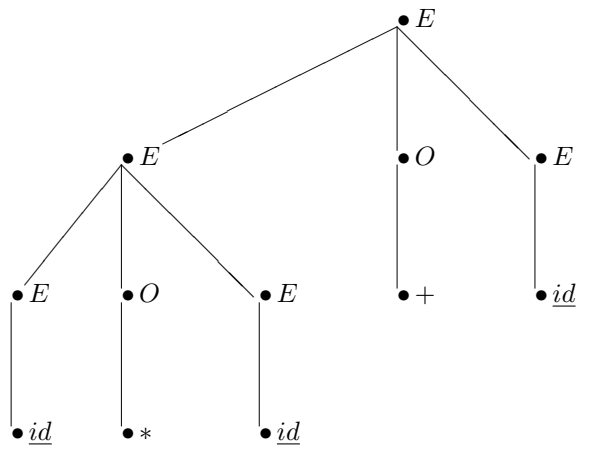
Consideremos la siguiente gramática incontextual G para generar expresiones aritméticas en donde aparezcan variables y las operaciones $+, -, *$ y $/$. Como estamos en la fase de análisis sintáctico del compilador, representamos a las variables por la categoría sintáctica "identificador", que denotamos 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. $E \longrightarrow (E)$

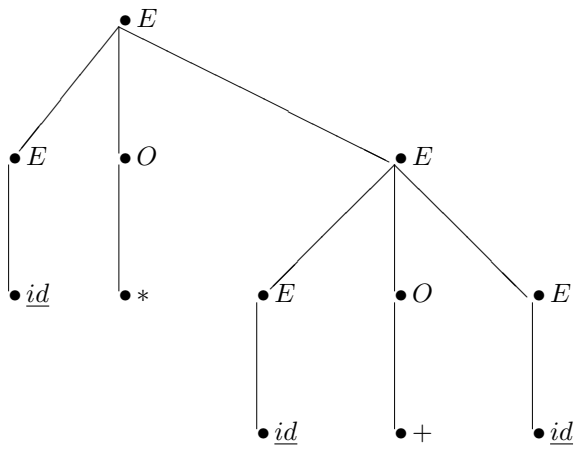
Ejemplo 1

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

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.

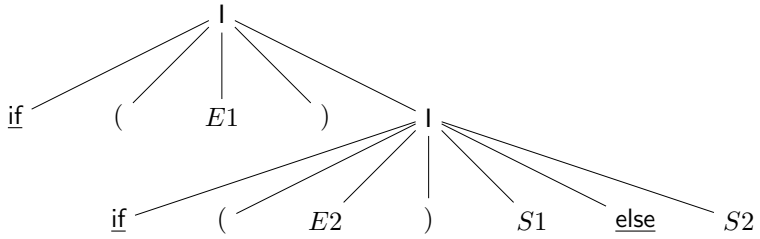
1. $I \longrightarrow \underline{if}\,(E)\,I\,\underline{else}\,I$
2. $I \longrightarrow \underline{if}\,(E)\,I$

La gramática es ambigua, porque una instrucción de la forma

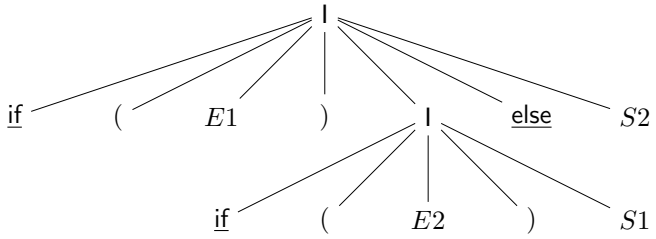
$\underline{if}\,(E1)\,\underline{if}\,(E2)\,S1\,\underline{else}\,S2$

(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.

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.

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.

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

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

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

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

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

Siguientes(X) $= \{a \in \Sigma :$ existen palabras α, β 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

Primeros$(\lambda \cdot$ Siguientes$(X))$ = Primeros(Siguientes$(X))$ = Siguientes(X),

ya que todo elemento de Siguientes(X) es un símbolo terminal, y para todo símbolo terminal t se tiene que 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 Siguientes(X). Entonces, para todo símbolo $t \in$ Siguientes(X), tendremos que $X \longrightarrow \lambda \in$ 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;
}

Introducción

En la clase de hoy, mostraremos dos reglas de eliminación del indeterminismo que nos permiten transformar muchas gramáticas no ambiguas en gramáticas LL(1).

Vamos a empezar recordando el algoritmo que vimos en la última clase para diseñar un analizador sintáctico.

Algoritmo para diseñar un analizador sintáctico

(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 se pueda programar.

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

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

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), que estudiamos en la última clase.

Recordemos el concepto de gramática LL(1).

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

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

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

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

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

Siguientes(X) $= \{a \in \Sigma :$ existen palabras α, β 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.

Reglas para transformar una gramática no ambigua en una gramática LL(1)

Dos gramáticas G_1 y G_2 son equivalentes, si generan el mismo lenguaje, es decir, si $L(G_1) = L(G_2)$.

Las dos siguientes reglas que veremos nos permiten transformar en muchos casos una gramática no ambigua en una gramática LL(1) equivalente.

Con dichas reglas se pretende eliminar el indeterminismo en el autómata con pila asociado a la gramática no ambigua de la que partimos.

Las dos reglas que veremos son las llamadas regla de factorización y regla de recursión.

Recuérdese que, para simplificar la notación, si $A \longrightarrow \alpha_1$, ..., $A \longrightarrow \alpha_n$ son reglas de una gramática incontextual con una misma parte izquierda, escribiremos $A \longrightarrow \alpha_1 | \ldots | \alpha_n$.

Regla de Factorización

Si tenemos producciones en la gramática de la forma $A \longrightarrow \alpha\beta_1 | \ldots | \alpha\beta_n$ donde $\alpha \neq \lambda$ y $n \geq 2$, reemplazar estas producciones por

$A \longrightarrow \alpha A'$,

$A' \longrightarrow \beta_1 | \ldots | \beta_n$

donde A' es una nueva variable.

Obsérvese que la aplicación de la Regla de Factorización no cambia el lenguaje generado por la gramática, ya que los dos bloques de producciones generan el lenguaje de la expresión regular $\alpha \cdot (\beta_1 \cup \cdots \cup \beta_n)$.

Regla de Recursión

Si tenemos producciones en la gramática de la forma $A \longrightarrow A\alpha_1|\ldots|A\alpha_n|\beta_1|\ldots|\beta_m$ donde $n \geq 1$ y los β_i no comienzan por A, reemplazar estas producciones por

$A \longrightarrow \beta_1 A'|\ldots|\beta_m A'$,

$A' \longrightarrow \alpha_1 A'|\ldots|\alpha_n A'|\lambda$

donde A' es una nueva variable.

Obsérvese que la aplicación de la Regla de Recursión no cambia el lenguaje generado por la gramática, ya que los dos bloques de producciones generan el lenguaje de la expresión regular $(\beta_1 \cup \cdots \cup \beta_m) \cdot (\alpha_1 \cup \cdots \cup \alpha_n)^*$.

Ejemplo 1

Consideremos la siguiente gramática incontextual G para diseñar una calculadora de dígitos decimales, donde E es el símbolo inicial.

1. $E \longrightarrow T$
2. $E \longrightarrow TOE$
3. $T \longrightarrow A$
4. $T \longrightarrow TPA$
5. $O \longrightarrow +$
6. $O \longrightarrow -$
7. $P \longrightarrow *$
8. $P \longrightarrow /$
9. $A \longrightarrow \underline{int}$
10. $A \longrightarrow \underline{float}$

Ejemplo 1

En G, la variable E genera de manera unívoca una suma/resta de términos y la variable T genera también de manera unívoca un producto/división de factores, que pueden ser o bien números enteros (tipo $\underline{int}$) o bien números decimales (tipo $\underline{float}$). Por tanto, G no es ambigua.

Sin embargo, G no es LL(1), ya que por ejemplo tenemos que las producciones $1, 2 \in TABLA[E, \underline{int}]$.

Queremos transformar entonces la gramática G en una gramática LL(1) equivalente. Para ello, utilizamos las reglas de factorización y recursión.

Ejemplo 1

Aplicando la regla de factorización, reemplazamos las producciones $E \longrightarrow T$ y $E \longrightarrow TOE$ por las producciones $E \longrightarrow TX$, $X \longrightarrow \lambda$ y $X \longrightarrow OE$.

Y aplicando la la regla de recursión, reemplazamos las producciones $T \longrightarrow A$ y $T \longrightarrow TPA$ por las producciones $T \longrightarrow AY$, $Y \longrightarrow PAY$ e $Y \longrightarrow \lambda$.

Ejemplo 1

Por tanto, obtenemos la siguiente gramática G' equivalente a G:

1. $E \longrightarrow TX$
2. $X \longrightarrow \lambda$
3. $X \longrightarrow OE$
4. $T \longrightarrow AY$
5. $Y \longrightarrow PAY$
6. $Y \longrightarrow \lambda$
7. $O \longrightarrow +$
8. $O \longrightarrow -$
9. $P \longrightarrow *$
10. $P \longrightarrow /$
11. $A \longrightarrow \underline{int}$
12. $A \longrightarrow \underline{float}$

Ejemplo 1

La tabla de análisis de G' es la siguiente:

	$+$	$-$	$*$	$/$	int	float
E					1	1
X	3	3				
T					4	4
Y	6	6	5	5		
O	7	8				
P			9	10		
A					11	12

Ejemplo 1

Obsérvese que de la derivación

$TX \Rightarrow^4 AYX \Rightarrow^{11} \underline{int}\,YX$

se deduce que $\underline{int} \in$ Primeros(TX) y, por tanto, la producción $1 \in$ TABLA$[S, \underline{int}]$.

Por otra parte, de la derivación

$TX \Rightarrow^4 AYX \Rightarrow^{12} \underline{float}\,YX$

se deduce que $\underline{float} \in$ Primeros(TX) y, por tanto, la producción $1 \in$ TABLA$[S, \underline{float}]$.

Ejemplo 1

Obsérvese que de la derivación

$OX \Rightarrow^7 +E$

se deduce que $+ \in$ Primeros(OX) y, por tanto, la producción $3 \in$ TABLA$[X, +]$.

Por otra parte, de la derivación

$OX \Rightarrow^8 -E$

se deduce que $- \in$ Primeros(OX) y, por tanto, la producción $3 \in$ TABLA$[X, -]$.

Ejemplo 1

Obsérvese que de la derivación

$AY \Rightarrow^7 \underline{int}Y$

se deduce que $\underline{int} \in$ Primeros(AY) y, por tanto, la producción $4 \in$ TABLA$[T, \underline{int}]$.

Análogamente, tenemos que la producción $4 \in$ TABLA$[T, \underline{float}]$.

Ejemplo 1

Obsérvese que de la derivación

$PAY \Rightarrow^9 *AY$

se deduce que $* \in$ Primeros(PAY) y, por tanto, la producción $5 \in$ TABLA$[Y, *]$.

Análogamente, tenemos que la producción $5 \in$ TABLA$[T, /]$.

Por otra parte, es claro que por las producciones 8-12, se deduce directamente que $7 \in$ TABLA$[O, +]$, $8 \in$ TABLA$[O, -]$, $9 \in$ TABLA$[P, *]$, $10 \in$ TABLA$[P, /]$, $11 \in$ TABLA$[A, \underline{int}]$ y $12 \in$ TABLA$[A, \underline{float}]$.

Ejemplo 1

Obsérvese que Siguientes$(X) = \emptyset$. Por tanto, la producción 2 no aparece en la tabla de análisis.

Y de las derivaciones

(1) $E \Rightarrow^1 TX \Rightarrow^3 TOE \Rightarrow^4 AYOE \Rightarrow^7 AY + E$

(2) $E \Rightarrow^1 TX \Rightarrow^3 TOE \Rightarrow^4 AYOE \Rightarrow^8 AY - E$

se deduce que $+, - \in$ Siguientes(Y) y, por tanto, la producción 6 pertenece a TABLA$[Y, +]$ y a TABLA$[Y, -]$.

Ejemplo 1

Por tanto, G' es LL(1), ya que su tabla de análisis no tiene conflictos.

Siguiendo entonces el algoritmo que mostramos en la última clase de teoría, el autómata con pila asociado a G se puede programar utilizando la tabla de análisis de G'.

Ejemplo 2

Consideramos la siguiente gramática G, que genera declaraciones de variables en C cuyos tipos son *integer* y *float*.Como estamos en la fase de análisis sintáctico del compilador, representamos a las variables por la categoría sintáctica "identificador", que denotamos por $\underline{id}$. Y representamos por $\underline{int}$ al tipo entero, y por $\underline{float}$ al tipo float. La gramática G está entonces definida por las siguientes producciones:

1. $S \longrightarrow D\,;\,S$
2. $S \longrightarrow D\,;$
3. $D \longrightarrow TV$
4. $T \longrightarrow \underline{int}$
5. $T \longrightarrow \underline{float}$
6. $V \longrightarrow V, \underline{id}$
7. $V \longrightarrow \underline{id}$

Ejemplo 2

La gramática G no es LL(1), porque hay conflictos al construir su tabla de análisis. Por una parte, como $\underline{int} \in$ Primeros(D), se tiene que las producciones 1 y 2 pertenecen a TABLA[S,$\underline{int}$]. Análogamente, las producciones 1 y 2 pertenecen a TABLA[S,$\underline{float}$].

Transformamos entonces la gramática G en una gramática LL(1) equivalente, aplicando las reglas de factorización y recursión.

Aplicando la regla de factorización, reemplazamos las producciones 1 y 2 de G por las producciones:

$S \longrightarrow D\,;\,S'$

$S' \longrightarrow S$

$S' \longrightarrow \lambda$

Ejemplo 2

Y aplicando la regla de recursión, reemplazamos las producciones 6 y 7 de G por:

$V \longrightarrow \underline{id}\,V'$

$V' \longrightarrow, \underline{id}\,V'$

$V' \longrightarrow \lambda$

Obtenemos entonces la gramática G' equivalente a G dada por las siguientes producciones:

Ejemplo 2

1. $S \longrightarrow D\,;\,S'$
2. $S' \longrightarrow S$
3. $S' \longrightarrow \lambda$
4. $D \longrightarrow TV$
5. $T \longrightarrow \underline{int}$
6. $T \longrightarrow \underline{float}$
7. $V \longrightarrow \underline{id}\,V'$
8. $V' \longrightarrow, \underline{id}\,V'$
9. $V' \longrightarrow \lambda$

La tabla de análisis para G' es entonces la siguiente:

Ejemplo 2

TABLA	id	int	float	,	;
S		1	1		
S'		2	2		
D		4	4		
T		5	6		
V	7				
V'				8	9

Obsérvese que de la derivación

$D \Rightarrow^4 TV \Rightarrow^5 \underline{int}\, V$

se deduce que $\underline{int} \in$ Primeros(D) y, por tanto, la producción $1 \in$ TABLA$[S, \underline{int}]$.

Ejemplo 2

Análogamente, de la derivación

$D \Rightarrow^4 TV \Rightarrow^6 \underline{float}\, V$

se deduce que $\underline{float} \in$ Primeros(D) y, por tanto, la producción $1 \in$ TABLA$[S, \underline{float}]$.

De la derivación

$S \Rightarrow^1 D\,; S' \Rightarrow^4 TV\,; S' \Rightarrow^5 \underline{int}\, V\,; S'$

se deduce que $\underline{int} \in$ Primeros(S) y, por tanto, la producción $2 \in$ TABLA$[S', \underline{int}]$.

Y de la derivación

$S \Rightarrow^1 D\,; S' \Rightarrow^4 TV\,; S' \Rightarrow^6 \underline{float}\, V\,; S'$

se deduce que $\underline{float} \in$ Primeros(S) y, por tanto, la producción $2 \in$ TABLA$[S', \underline{float}]$.

Ejemplo 2

Por otra parte, tenemos que Primeros$(T) = \{\underline{int}, \underline{float}\}$ por las producciones 5 y 6. Por tanto, la producción 4 pertenece a TABLA$[D, \underline{int}]$ y pertenece también a TABLA$[D, \underline{float}]$.

Tenemos que la producción 7 pertenece a TABLA$[V, \underline{id}]$, ya que $\underline{id}$ es el primer símbolo de la parte derecha de 7. Y análogamente, la producción 8 pertenece a TABLA$[V', ,]$, ya que , es el primer símbolo de la parte derecha de 8.

Por último, de la derivación

$S \Rightarrow^1 D\,; S' \Rightarrow^4 TV\,; S' \Rightarrow^7 T\,\underline{id}\,V'\,; S'$

deducimos que $; \in$ Siguientes(V') y, por tanto, la producción $9 \in$ TABLA$[V', ;]$.

Ejemplo 3

Queremos escribir un programa para reconocer expresiones aritméticas de un lenguaje de programación, como C o Java. Este programa es una parte importante del analizador sintáctico del compilador. Para simplificar la exposición, suponemos que en las expresiones aritméticas aparecen únicamente las operaciones suma y producto. Consideramos entonces la siguiente gramática no ambigua G para generar expresiones aritméticas:

1. $E \longrightarrow E + T$
2. $E \longrightarrow T$
3. $T \longrightarrow T * F$
4. $T \longrightarrow F$
5. $F \longrightarrow (E)$
6. $F \longrightarrow \underline{int}$
7. $F \longrightarrow \underline{float}$
8. $F \longrightarrow \underline{id}$
9. $F \longrightarrow \underline{id}(E)$

Ejemplo 3

Hemos incluido la producción 9 para poder tratar llamadas a funciones. Se trata de una gramática standard que se utiliza en el diseño de compiladores para tratar las expresiones aritméticas. Se puede comprobar fácilmente que la gramática G no es ambigua, ya que con la variable E se genera una suma de términos de manera unívoca, con la variable T se genera un producto 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, elementos del tipo integer, elementos del tipo float, identificadores o llamadas a funciones.

Por ejemplo, tenemos la siguiente derivación para la palabra $x = (\underline{id} * \underline{int} + \underline{float}) * \underline{id}$.

Ejemplo 3

$E \Rightarrow^2 T \Rightarrow^3 T * F \Rightarrow^8 T * \underline{id} \Rightarrow^4 F * \underline{id}$
$\Rightarrow^5 (E) * \underline{id} \Rightarrow^1 (E + T) * \underline{id} \Rightarrow^4 (E + F) * \underline{id}$
$\Rightarrow^7 (E + \underline{float}) * \underline{id} \Rightarrow^2 (T + \underline{float}) * \underline{id} \Rightarrow^3 (T * F + \underline{float}) * \underline{id}$
$\Rightarrow^6 (T * \underline{int} + \underline{float}) * \underline{id} \Rightarrow^4 (F * \underline{int} + \underline{float}) * \underline{id}$
$\Rightarrow^8 (\underline{id} * \underline{int} + \underline{float}) * \underline{id}$.

Sin embargo, G no es LL(1). Por ejemplo, tenemos que las producciones 1 y 2 están en TABLA$[E, \underline{id}]$, ya que $\underline{id} \in$ Primeros$(E) \cap$ Primeros(T).

Queremos transformar entonces la gramática G en una gramática LL(1) equivalente. Para ello, utilizamos las reglas de factorización y recursión.

Ejemplo 3

Aplicando la regla de factorización a las producciones 8 y 9, reemplazamos dichas producciones por las siguientes:

$F \longrightarrow \underline{id}\, F'$
$F' \longrightarrow (E)$
$F' \longrightarrow \lambda$

Aplicando ahora la regla de recursión a las producciones 1 y 2, reemplazamos dichas producciones por las siguientes:

$E \longrightarrow T\, E'$
$E' \longrightarrow +\, T\, E'$
$E' \longrightarrow \lambda$

Ejemplo 3

Por último, aplicamos la regla de recursión a las producciones 3 y 4 reemplazando dichas producciones por las siguientes:

$T \longrightarrow F\, T'$
$T' \longrightarrow *\, F\, T'$
$T' \longrightarrow \lambda$

Obtenemos entonces la gramática LL(1) G' equivalente a G dada por las siguientes producciones:

Ejemplo 3

1. $E \longrightarrow T\,E'$.
2. $E' \longrightarrow +T\,E'$.
3. $E' \longrightarrow \lambda$.
4. $T \longrightarrow FT'$.
5. $T' \longrightarrow *FT'$.
6. $T' \longrightarrow \lambda$.
9. $F \longrightarrow (E)$.
10. $F \longrightarrow \underline{int}$.
11. $F \longrightarrow \underline{float}$.
12. $F \longrightarrow \underline{id}\,F'$.
13. $F' \longrightarrow \lambda$.
14. $F' \longrightarrow (E)$.

Ejemplo 3

Se tiene entonces que G' es una gramática equivalente a G, ya que la aplicación de las reglas de factorización y recursión no cambia el lenguaje generado por la gramática de partida. Es decir, el lenguaje generado por G' es el lenguaje de las expresiones aritméticas. Se tiene además que G' es LL(1), y por tanto podemos utilizar el algoritmo para programar el autómata con pila asociado a una gramática LL(1) visto anteriormente. El programa resultante es entonces el analizador sintáctico para expresiones aritméticas. Por tanto, dicho programa recibe como entrada una tira de símbolos y determina si la entrada es una expresión aritmética.

Introducción

En la primera parte de la clase de hoy, terminaremos el ejemplo del diseño de un programa para reconocer expresiones aritméticas, que empezamos en la última clase.

Y en la segunda parte de la clase, estudiaremos la fase del análisis semántico del compilador, y mostraremos el tipo de gramáticas que se utilizan en esta fase, que son las llamadas "gramáticas de atributos".

Empezaremos recordando las reglas de factorización y recursión, las cuales nos permiten transformar muchas gramáticas incontextuales no ambiguas en gramáticas LL(1).

Regla de Factorización

Si tenemos producciones en la gramática de la forma $A \longrightarrow \alpha\beta_1|\ldots|\alpha\beta_n$ donde $\alpha \neq \lambda$ y $n \geq 2$, reemplazar estas producciones por

$A \longrightarrow \alpha A'$,

$A' \longrightarrow \beta_1|\ldots|\beta_n$

donde A' es una nueva variable.

Obsérvese que la aplicación de la Regla de Factorización no cambia el lenguaje generado por la gramática, ya que los dos bloques de producciones generan el lenguaje de la expresión regular $\alpha \cdot (\beta_1 \cup \cdots \cup \beta_n)$.

Regla de Recursión

Si tenemos producciones en la gramática de la forma $A \longrightarrow A\alpha_1|\ldots|A\alpha_n|\beta_1|\ldots|\beta_m$ donde $n \geq 1$ y los β_i no comienzan por A, reemplazar estas producciones por

$A \longrightarrow \beta_1 A'|\ldots|\beta_m A'$,

$A' \longrightarrow \alpha_1 A'|\ldots|\alpha_n A'|\lambda$

donde A' es una nueva variable.

Obsérvese que la aplicación de la Regla de Recursión no cambia el lenguaje generado por la gramática, ya que los dos bloques de producciones generan el lenguaje de la expresión regular $(\beta_1 \cup \cdots \cup \beta_m) \cdot (\alpha_1 \cup \cdots \cup \alpha_n)^*$.

Ejemplo

Queremos escribir un programa para reconocer expresiones aritméticas de un lenguaje de programación, como C o Java. Este programa es una parte importante del analizador sintáctico del compilador. Se trata de un programa complejo que no se puede diseñar directamente. Para poderlo diseñar, hay que utilizar las técnicas que hemos estudiado en este tema. Para simplificar la exposición, suponemos que en las expresiones aritméticas aparecen únicamente las operaciones suma y producto. Consideramos entonces la siguiente gramática no ambigua G para generar expresiones aritméticas:

Ejemplo

1. $E \longrightarrow E + T$
2. $E \longrightarrow T$
3. $T \longrightarrow T * F$
4. $T \longrightarrow F$
5. $F \longrightarrow (E)$
6. $F \longrightarrow \underline{int}$
7. $F \longrightarrow \underline{float}$
8. $F \longrightarrow \underline{id}$
9. $F \longrightarrow \underline{id}(E)$

Ejemplo

Hemos incluido la producción 9 para poder tratar llamadas a funciones. Se trata de una gramática standard que se utiliza en el diseño de compiladores para tratar las expresiones aritméticas. Se puede comprobar fácilmente que la gramática G no es ambigua, ya que con la variable E se genera una suma de términos de manera unívoca, con la variable T se genera un producto 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, elementos del tipo integer, elementos del tipo float, identificadores o llamadas a funciones.

Por ejemplo, tenemos la siguiente derivación para la palabra $x = (\underline{id} * \underline{int} + \underline{float}) * \underline{id}$.

Ejemplo

$E \Rightarrow^2 T \Rightarrow^3 T * F \Rightarrow^8 T * \underline{id} \Rightarrow^4 F * \underline{id}$
$\Rightarrow^5 (E) * \underline{id} \Rightarrow^1 (E + T) * \underline{id} \Rightarrow^4 (E + F) * \underline{id}$
$\Rightarrow^7 (E + \underline{float}) * \underline{id} \Rightarrow^2 (T + \underline{float}) * \underline{id} \Rightarrow^3 (T * F + \underline{float}) * \underline{id}$
$\Rightarrow^6 (T * \underline{int} + \underline{float}) * \underline{id} \Rightarrow^4 (F * \underline{int} + \underline{float}) * \underline{id}$
$\Rightarrow^8 (\underline{id} * \underline{int} + \underline{float}) * \underline{id}$.

Sin embargo, G no es LL(1). Por ejemplo, tenemos que las producciones 1 y 2 están en TABLA$[E, \underline{id}]$, ya que $\underline{id} \in$ Primeros$(E) \cap$ Primeros(T).

Queremos transformar entonces la gramática G en una gramática LL(1) equivalente. Para ello, en la última clase, utilizando las reglas de factorización y recursión, obtuvimos la siguiente gramática G' equivalente a G:

Ejemplo

1. $S \longrightarrow E\$$.
2. $E \longrightarrow T\, E'$.
3. $E' \longrightarrow +T\, E'$.
4. $E' \longrightarrow \lambda$.
5. $T \longrightarrow FT'$.
6. $T' \longrightarrow *FT'$.
7. $T' \longrightarrow \lambda$.
8. $F \longrightarrow (E)$.
9. $F \longrightarrow \underline{id}\, F'$.
10. $F' \longrightarrow \lambda$.
11. $F' \longrightarrow (E)$.
12. $F \longrightarrow \underline{int}$.
13. $F \longrightarrow \underline{float}$.

Ejemplo

Introducimos la producción 1 de G' para poner el carácter \$ al final de la palabra de entrada. Se tiene entonces que G' es una gramática equivalente a G, ya que la aplicación de las reglas de factorización y recursión no cambia el lenguaje generado por la gramática de partida. Es decir, el lenguaje generado por G' es el lenguaje de las expresiones aritméticas. Vamos a demostrar entonces que G' es LL(1). Para ello, calculamos en primer lugar los Primeros y los Siguientes de las variables de G'. Se deduce directamente de las producciones de G' que

Primeros(S) = Primeros(E) = Primeros(T) = Primeros(F) = $\{(, \underline{id}, \underline{int}, \underline{float}\}$,
Primeros$(E') = \{+, \lambda\}$,
Primeros$(T') = \{*, \lambda\}$,
Primeros$(F') = \{), \lambda\}$.

Ejemplo

Tenemos que
Siguientes(E) = Siguientes$(E') = \{\$,)\}$,
debido a las siguientes derivaciones:
$S \Rightarrow^1 E\$ \Rightarrow^2 TE'\$$,
$S \Rightarrow^1 E\$ \Rightarrow^2 TE'\$ \Rightarrow^4 T\$ \Rightarrow^5 FT'\$ \Rightarrow^7 F\$ \Rightarrow^8 (E)\$ \Rightarrow^2 (TE')\$$.

Ejemplo

Tenemos que

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

debido a las siguientes derivaciones:

$S \Rightarrow^1 E\$ \Rightarrow^2 TE'\$ \Rightarrow^4 T\$ \Rightarrow^5 FT'\$$,

$S \Rightarrow^1 E\$ \Rightarrow^2 TE'\$ \Rightarrow^4 T\$ \Rightarrow^5 FT'\$ \Rightarrow^7 F\$ \Rightarrow^8 (E)\$ \Rightarrow^2 (TE')\$ \Rightarrow^4 (T)\$ \Rightarrow^5 (FT')\$$,

$S \Rightarrow^1 E\$ \Rightarrow^2 TE'\$ \Rightarrow^3 T + TE'\$ \Rightarrow^5 FT' + TE`\$$.

Ejemplo

Y tenemos que

Siguientes(F) = Siguientes$(F') = \{\$,), +, *\}$,

debido a las siguientes derivaciones:

$S \Rightarrow^1 E\$ \Rightarrow^2 TE'\$ \Rightarrow^4 T\$ \Rightarrow^5 FT'\$ \Rightarrow^7 F\$ \Rightarrow^9 \underline{id}\, F'\$$,

$S \Rightarrow^1 E\$ \Rightarrow^2 TE'\$ \Rightarrow^4 T\$ \Rightarrow^5 FT'\$ \Rightarrow^7 F\$ \Rightarrow^8 (E)\$ \Rightarrow^2 (TE')\$ \Rightarrow^4 (T)\$ \Rightarrow^5 (FT')\$ \Rightarrow^7 (F)\$ \Rightarrow^9 (\underline{id}\, F')\$$,

$S \Rightarrow^1 E\$ \Rightarrow^2 TE'\$ \Rightarrow^3 T + TE'\$ \Rightarrow^5 FT' + TE`\$ \Rightarrow^7 F + TE`\$ \Rightarrow^9 \underline{id}\, F' + TE`\$$,

$S \Rightarrow^1 E\$ \Rightarrow^2 TE'\$ \Rightarrow^3 T + TE'\$ \Rightarrow^5 FT' + TE`\$ \Rightarrow^6 F * FT' + TE'\$ \Rightarrow^9 \underline{id}\, F' * FT' + TE'\$$.

La tabla de análisis de G' es entonces la siguiente:

Ejemplo

TABLA	$\underline{id}$	$\underline{int}$	$\underline{float}$	$+$	$*$	$($	$)$	\$
S	1	1	1			1		
E	2	2	2			2		
E'				3			4	4
T	5	5	5			5		
T'				7	6		7	7
F	9	12	13			8		
F'				10	10	11	10	10

Ejemplo

Obsérvese que:

la producción 4 pertenece a TABLA$[E',)]$ y a TABLA$[E', \$]$, porque Siguientes$(E') = \{\$,)\}$,

la producción 7 pertenece a TABLA$[T', +]$, a TABLA$[T',)]$ y a TABLA$[T', \$]$, porque Siguientes$(T') = \{\$,), +\}$,

y la producción 10 pertenece a TABLA$[F', +]$, a TABLA$[F', *]$, a TABLA$[F',)]$ y a TABLA$[F', \$]$, porque Siguientes$(F') = \{\$,), +, *\}$.

Como no hay conflictos en la tabla de análisis de G', tenemos que G' es LL(1). Por tanto, el autómata con pila asociado a G' se puede programar, utilizando el algoritmo que vimos para programar el autómata con pila asociado a una gramática LL(1), es decir, el algoritmo que mostramos a continuación. El programa asociado al autómata con pila de G' es entonces el analizador sintáctico para reconocer expresiones aritméticas.

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 '$' en la pila
  (2) Aplicar la transicion ((q0, λ, λ), (f, S))
  (3) while ((tope de la pila ! = '$' || caracter de entrada ! = '$')
              && b)
      { if (tope de la pila == 'a' && caracter de entrada == 'a' )
        {aplicar la transicion ((f, a, a), (f, λ)); leer siguiente caracter;}
         else if ( tope de la pila == 'X' && caracter de entrada == 'a'
                  &&TABLA[X, a] == X ⟶ X1...Xn)
              aplicar la transicion ((f, λ, X), (f, X1...Xn));
              else b = false; }
  (4) if (tope de la pila == '$' && caracter de entrada == '$')
        return true; else return false;
}
```

Análisis semántico

Como ya hemos indicado anteriormente, el analizador sintáctico verifica que la palabra que recibe como entrada cumple las reglas del lenguaje de programación. Y si es así, proporciona como representación del proceso de análisis realizado el árbol de derivación de la palabra que recibe como entrada, el cual es utilizado por el analizador semántico para verificar las restricciones semánticas del lenguaje de programación.

Además, en la fase del análisis semántico se realiza la construcción de la llamada tabla de símbolos del compilador, en la cual se almacena la información necesaria sobre los identificadores que se utilizan en el programa: nombre, tipo, valor, dirección de memoria, categoría (si es variable, constante, método, función, ...), tamaño (si se trata de un vector o una matriz),etc. El analizador semántico utiliza entonces el árbol de derivación de la palabra de entrada y la tabla de símbolos del compilador.

Gramáticas de atributos

Para realizar el análisis semántico, se utilizan las llamadas gramáticas de atributos, las cuales son un refinamiento de las gramáticas incontextuales que hemos estudiado hasta ahora.

Queremos dar significado a los símbolos de una gramática incontextual. Para ello, asignamos atributos a los símbolos de la gramática.

En general, los atributos representan propiedades del lenguaje de programación, como pueden ser el tipo de datos de una variable, el valor de una expresión o la ubicación de una variable en memoria.

Gramáticas de atributos

Entonces, para poder realizar el análisis semántico hay que manipular los atributos asignando a las reglas de la gramática incontextual código de un lenguaje de programación. A dichos códigos se les llama acciones semánticas.

Definimos entonces una gramática de atributos como una gramática incontextual $G = (V, \Sigma, P, S)$, en la cual hemos definido atributos para un subconjunto de símbolos de $V \cup \Sigma$ y hemos definido acciones semánticas para las producciones de P.

Observaciones

(1) Los atributos que aparezcan en una acción semántica asociada a una regla de la gramática deben ser atributos de los símbolos de esa regla de la gramática.

(2) Los símbolos de una regla de una gramática de atributos no pueden estar repetidos. Para ello, hay que renombrar los símbolos que se repitan en una regla de la gramática.

El punto (2) es necesario, porque si no se renombran los símbolos en las reglas de una gramática, los atributos pueden quedar indefinidos.

Observaciones

Entonces, el análisis semántico para una expresión que estemos considerando se efectúa recorriendo el árbol de derivación de la expresión y ejecutando las acciones semánticas correspondientes.

Si a es un atributo asignado a un símbolo X de una gramática, nos referiremos a él como $X.a$.

Ejemplo

Consideremos la siguiente gramática incontextual G para declarar variables de tipo entero o de tipo real.

1. $S \to TX$
2. $T \to \underline{int}$
3. $T \to \underline{float}$
4. $X \to L\,;$
5. $L \to \underline{id}, L$
6. $L \to \underline{id}$

Asignamos el atributo _tipo a los símbolos T, X, L e $\underline{id}$.

Asignamos entonces las siguientes acciones semánticas a las reglas de G:

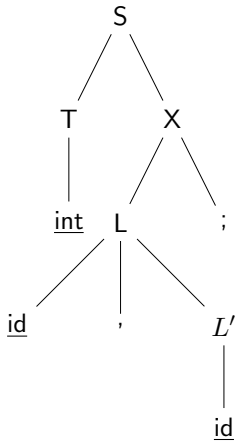
Ejemplo

Regla gramatical	Acción semántica
1. $S \to TX$	X.tipo $=$ T.tipo
2. $T \to \underline{int}$	T.tipo $=$ integer
3. $T \to \underline{float}$	T.tipo $=$ real
4. $X \to L\,;$	L.tipo $= X$.tipo
5. $L \to \underline{id}, L'$	$\underline{id}$.tipo $= L$.tipo; L'.tipo $= L$.tipo
6. $L \to \underline{id}$	$\underline{id}$.tipo $= L$.tipo

Consideremos ahora la declaración de variables $\underline{int}\ \underline{id}\ ,\ \underline{id};$

Para hacer entonces el análisis semántico de esta declaración, en primer lugar se considera el árbol de derivación de la declaración:

Ejemplo



Ejemplo

Entonces, por la acción semántica asociada a la regla 2, el analizador semántico deduce que T.tipo $= \underline{integer}$. Ahora, por la acción semántica asociada a la regla 1, deduce que X.tipo $= \underline{integer}$. A continuación, por la acción semántica asociada a la regla 4, deduce que L.tipo $= \underline{integer}$. Entonces, por la acción semántica asociada a la regla 5, deduce que el tipo del primer identificador es un entero. Aplicando de nuevo la acción semántica asociada a la regla 5, deduce que L'.tipo $= \underline{integer}$. Por último, aplicando la acción semántica asociada a la regla 6, deduce que el tipo del segundo identificador de la declaración es también un entero.

Tipos de atributos

Los tipos principales de atributos que se utilizan en el análisis semántico son los llamados atributos sintetizados y atributos heredados.

Un atributo es sintetizado, si su valor se calcula a partir de los valores de los atributos de sus nodos hijos en el árbol de derivación.

Y un atributo es heredado, si su valor se calcula a partir de los valores de los atributos de sus nodos hermanos o nodos padres en el árbol de derivación.

En el ejemplo anterior, tenemos que el atributo .tipo para T es sintetizado, y el atributo .tipo para L, X e $\underline{id}$ es heredado.

Ejemplo

Consideremos la siguiente gramática incontextual G para generar expresiones aritméticas:

1. $E \to E + T$
2. $E \to T$
3. $T \to T * F$
4. $T \to F$
5. $F \to (E)$
6. $F \to \underline{id}$
7. $F \to \underline{int}$
8. $F \to \underline{float}$

Asignamos el atributo .valor a los símbolos E, T, F, $\underline{id}$, $\underline{int}$, $\underline{float}$.

Ejemplo

Asignamos entonces las siguientes acciones semánticas a las reglas de G:

Regla gramatical	Acción semántica
1. $E \to E' + T$	E.valor $= E'$.valor $+ T$.valor
2. $E \to T$	E.valor $= T$.valor
3. $T \to T' * F$	T.valor $= T'$.valor $* F$.valor
4. $T \to F$	T.valor $= F$.valor
5. $F \to (E)$	F.valor $= E$.valor
6. $F \to \underline{id}$	F.valor $= \underline{id}$.valor
7. $F \to \underline{int}$	F.valor $= \underline{int}$.valor
8. $F \to \underline{\underline{float}}$	F.valor $= \underline{\underline{float}}$.valor

Ejemplo

En este caso, los atributos E.valor, T.valor, F.valor, $\underline{id}$.valor, $\underline{int}$.valor y $\underline{\underline{float}}$.valor son sintetizados.

Procediendo entonces como en el ejemplo anterior, el analizador semántico calculará el valor de una expresión aritmética, utilizando el árbol de derivación de la expresión y las acciones semánticas de la gramática.