TEMA 3

ANÁLISIS SINTÁCTICO

Índice

ESTE tema está dedicado a los métodos de análisis sintáctico que se usan habitualmente en el desarrollo de compiladores. En primer lugar se estudian los objetivos del análisis sintáctico. Seguidamente se presentan los conceptos básicos relativos a las gramáticas libres de contexto, que constituyen la herramienta teórica principal para esta etapa del compilador. Tras esto se describen los dos enfoques descentente y ascendente para la generación del árbol de análisis del programa fuente, mostrando en detalle distintos métodos junto con sus estrategias de manejo de errores sintácticos.

3.1. Objetivos del analizador sintáctico

El papel principal del analizador sintáctico es el de producir, a partir de una cadena de componentes léxicos, una salida que consistirá en:

- Un árbol sintáctico que se proporcionará a la siguiente etapa del front-end, en el caso de que la secuencia de tokens verifique la gramática libre de contexto del lenguaje fuente.
- Si la secuencia de tokens contiene errores sintácticos, es decir, alguna frase no se ajusta a la estructura sintáctica definida por la gramática del lenguaje fuente, el compilador generará un informe con la lista de errores detectados. Este informe deberá ser lo más claro y exacto posible.

El analizador sintáctico puede realizar otras tareas complementarias, como son:

- Completar la tabla de símbolos con información sobre los tokens.
- Realizar tareas correspondientes al análisis semántico como la verificación de tipos.
- Generar código intermedio de forma simultánea a la realización del análisis sintáctico.

Es importante tener en cuenta que algunas construcciones sintácticas no pueden ser especificadas con gramáticas libres de contexto, y por tanto no pueden ser tratadas en esta fase del compilador, sino que deben ser pospuestas a la fase de análisis semántico. Por ejemplo, el requisito de que los identificadores estén declarados antes de ser usados no se puede expresar con una gramática libre de contexto. Por tanto, el analizador sintáctico acepta secuencias de tokens que forman un superconjunto de las secuencias realmente válidas. En fases posteriores del compilador se debe inspeccionar la salida del analizador sintáctico para comprobar si verifica las reglas que éste no puede revisar.

3.1.1. Manejo y recuperación de errores sintácticos

Si el compilador tuviese que procesar programas correctos únicamente, su diseño e implementación se simplificaría enormemente. Sin embargo, los programadores cometen inevitablemente errores al escribir los programas, y muchos de estos se pueden clasificar como errores sintácticos. Por tanto, se espera que el compilador asista al programador en la localización de los errores y su reparación.

Curiosamente, la mayoría de las especificaciones de los lenguajes de programación no describen cómo debe responder el compilador ante la presencia de errores; el manejo de los mismos queda a la elección del diseñador del compilador. Por ello, la planificación de esta tarea desde el comienzo del desarrollo del compilador es de gran importancia.

Los errores *sintácticos* incluyen, por dar algunos ejemplos, el olvido de tokens separadores de sentencias[1], la falta de balanceo en el uso de tokens delimitadores de bloques[2], la ausencia de operadores aritméticos o lógicos en expresiones que los requieren, etc.

La precisión de los métodos de análisis sintáctico que se estudian en este tema permite que este tipo de errores se detecte de forma muy eficiente. En algunos métodos, la detección se produce tan pronto como es

[1] En el lenguaje C se emplea el carácter `;`.

[2] En el lenguaje C se emplean los caracteres `{` y `}`.

posible, es decir, en el momento en que la secuencia de tokens deja de ser *viable* en el sentido de que deja de verificar la gramática del lenguaje fuente.

La gestión de los errores en el análisis sintáctico debe procurar cumplir los siguientes objetivos:

- El informe de la presencia de errores debe ser claro y preciso.
- La recuperación de los errores debe ser rápida de modo que se pueda continuar el procesamiento de la entrada para detectar errores subsiguientes.
- Los programas correctos no deben sufrir un procesamiento lento a causa del mecanismo de detección de errores.

La técnica de recuperación de errores más sencilla y conocida es la denominada *recuperación en modo pánico*. Cuando se recibe un token que no concuerda con la especificación sintáctica del lenguaje, el analizador comienza a desechar tokens hasta que encuentra uno que forma parte de un conjunto de tokens especial denominado *conjunto de sincronización*. En este conjunto pueden encontrarse los tokens de finalización de estructuras sintácticas, como por ejemplo `;` o `end`. A partir de este punto, se retoma el análisis normal. El inconveniente principal que presenta esta técnica es que desecha una gran cantidad de tokens que no son analizados en busca de nuevos errores; no obstante, es muy sencilla de implementar y es una estrategia libre de bucles infinitos.

Otra técnica es la *recuperación a nivel de frase*. En esta técnica, cuando se descubre el error, se realiza una corrección local insertando una cadena que permita continuar con el análisis sintáctico. Por ejemplo, se podría insertar un `;` cuando encuentra una sentencia finalizada con un signo de puntuación incorrecto. Sin embargo, tiene como desventaja su dificultad para afrontar situaciones en las que el error se produjo antes del punto de detección. Por otro lado, se corre el riesgo de caer en bucles infinitos.

Una técnica muy atractiva desde el punto de vista formal es la que emplea *producciones de error*. Si la especificación de la gramática se ha hecho de forma correcta, y se conoce en qué puntos suelen estar los errores más frecuentes, se puede ampliar la gramática con reglas de producción que simulen la producción de errores. Así, si se produce un error contemplado por la gramática ampliada, el análisis podría seguir y el diagnóstico producido sería el adecuado.

Por último, la técnica de *corrección global* se asienta en algoritmos que calculan la distancia mínima de una cadena incorrecta a una cadena correcta en términos de cambios en la primera para convertirla en la segunda. Estos métodos son demasiado costosos aunque tienen mucho interés teórico. Por ejemplo, pueden utilizarse para realizar una evaluación de otras técnicas de recuperación de errores, o bien pueden ser usados localmente para encontrar cadenas de sustitución óptimas en una recuperación a nivel de frase.

3.2. Fundamentos teóricos del análisis sintáctico

En esta sección se estudian las herramientas teóricas usadas en la fase del análisis sintáctico. En concreto, se describen las características fundamentales de las gramáticas libres de contexto y de las máquinas abstractas que reconocen los lenguajes descritos por éstas, es decir, los autómatas de pila. También se analiza el problema de la ambigüedad en el análisis sintáctico.

3.2.1. Gramáticas libres de contexto

De entre las cuatro clases de gramáticas de la clasificación de Chomsky (véase el tema 1), el grupo más importante, desde el punto de vista de su aplicación en el desarrollo de compiladores, es el de las *gramáticas independientes o libres de contexto*. Las gramáticas de este tipo se pueden usar para expresar la mayoría de estructuras sintácticas de un lenguaje de programación.

Definición 3.1. *Una gramática $G = (V_N, V_T, S, P)$ se denomina **libre de contexto** si sus producciones tienen la forma $A \to \alpha$, siendo $A \in V_N$ y $\alpha \in (V_N \cup V_T)^*$.*

El conjunto V_T representa a los símbolos básicos del lenguaje descrito por la gramática. Habitualmente se denominan símbolos *terminales*. Se puede considerar que el término *nombre de token* es sinónimo de terminal, y cuando no hay lugar a la confusión, se puede emplear directamente la palabra token como equivalente a terminal.

El conjunto V_N representa a las *variables sintácticas* que denotan conjuntos de cadenas (sentencias o expresiones del lenguaje fuente, por ejemplo). Este conjunto impone una estructura jerárquica al lenguaje que es clave en el análisis sintáctico y la generación de código. El no terminal S denota el *símbolo inicial* de la gramática, y representa el conjunto de todas las cadenas del lenguaje descrito por la gramática.

Las producciones de la gramática están formadas por un no terminal llamado *cabeza* o *lado izquierdo* de la producción. Seguidamente se emplea el símbolo $\to$, que en algunos textos se sustituye con `::=`. El *cuerpo* o *lado derecho* de la regla consiste en cero o más terminales y no terminales que describen una forma posible de las cadenas representadas por el no terminal del lado izquierdo. El símbolo λ se emplea para representar un lado derecho vacío.

Definición 3.2. *Sea una gramática $G = (V_N, V_T, P, S)$. Se dice que la cadena α **deriva directamente** en la cadena β, denotándolo $\alpha \Rightarrow \beta$, si se puede escribir:*

$$\alpha = \delta A \mu \ , \ \beta = \delta \gamma \mu$$

siendo δ y $\mu \in (V_T \cup V_N)^*$, y $A \to \gamma \in P$. Si aplicamos repetidamente el concepto de derivación directa:

$$\alpha = \gamma_0 \Rightarrow \gamma_1 \Rightarrow \ldots \Rightarrow \gamma_n = \beta, \ n > 0$$

*obtenemos una secuencia que se denomina **derivación de longitud** n. Esta derivación se puede expresar como $\alpha \Rightarrow^+ \beta$. También se puede incluir el caso de la identidad ($n \geq 0$), denotándolo con $\alpha \Rightarrow^* \beta$.*

Definición 3.3. *Sea una gramática $G = (V_N, V_T, P, S)$. Para cualquier $A \in V_N$ y $\alpha \in (V_N \cup V_T)^*$ se dice que A **deriva** en α, denotado con $A \Rightarrow^* \alpha$, si existe una cadena de derivaciones de longitud cualquiera, incluso de longitud 0, desde A hasta α.*

Definición 3.4. *Sea una gramática $G = (V_N, V_T, P, S)$. Se denominan **formas sentenciales** de G a las cadenas de símbolos del conjunto:*

$$D(G) = \{\alpha \mid S \Rightarrow^* \alpha \ , \ \alpha \in (V_N \cup V_T)^*\}$$

Definición 3.5. *Una forma sentencial x tal que $x \in V_T^*$ se dice que es una **sentencia**.*

Definición 3.6. *El **lenguaje definido por una gramática** G, denotado $L(G)$, es el conjunto de sentencias que se pueden derivar partiendo del símbolo inicial de la gramática. Es decir:*

$$L(G) = \{x \mid S \Rightarrow^* x \ , \ x \in V_T^*\}$$

Si al realizar las derivaciones directas escogemos el no terminal que esté situado más a la izquierda en la forma sentencial, obtendremos una derivación directa *más a la izquierda*, que denotaremos con $\Rightarrow_{mi}$. De forma similar, si escogemos el que esté situado más a la derecha, obtendremos una derivación directa *más a la derecha*, que denotaremos $\Rightarrow_{md}$.

Definición 3.7. *Una **derivación más a la izquierda** de una forma sentencial α es una derivación que comienza con el símbolo inicial S, acaba en la forma sentencial, y en cada paso de derivación se realiza una derivación directa más a la izquierda, es decir, $S \Rightarrow_{mi}^{*} \alpha$. Una forma sentencial obtenida de este modo se denomina **forma sentencial izquierda**.*

*De forma similar se puede definir una **derivación más a la derecha** como aquella que comienza en S, acaba en α, y en cada paso de derivación se realiza una derivación directa más a la derecha, es decir, $S \Rightarrow_{md}^{*} \alpha$. Una forma sentencial obtenida de este modo se denomina **forma sentencial derecha**.*

Ejemplo 3.1. Disponemos de una gramática $G = (V_N, V_T, P, E)$ tal que $V_N = \{E\}$, $V_T = \{+, -, *, (,), \mathbf{id}\}$ y el conjunto de reglas P es el siguiente:

$$E \to E + E$$
$$E \to E * E$$
$$E \to -E$$
$$E \to (E)$$
$$E \to \mathbf{id}$$

Podemos obtener una derivación más a la izquierda de la sentencia $-(\mathbf{id} + \mathbf{id})$ de la siguiente forma:

$$E \Rightarrow_{mi} -E \Rightarrow_{mi} -(E) \Rightarrow_{mi} -(E + E) \Rightarrow_{mi} -(\mathbf{id} + E) \Rightarrow_{mi} -(\mathbf{id} + \mathbf{id})$$

Por otra parte, podemos obtener una derivación más a la derecha de la misma sentencia de la siguiente forma:

$$E \Rightarrow_{md} -E \Rightarrow_{md} -(E) \Rightarrow_{md} -(E + E) \Rightarrow_{md} -(E + \mathbf{id}) \Rightarrow_{md} -(\mathbf{id} + \mathbf{id})$$

Definición 3.8. *A una derivación más a la derecha como la siguiente:*

$$S \Rightarrow_{md} \gamma_1 \Rightarrow_{md} \gamma_2 \Rightarrow_{md} \dots \gamma_{n-1} \Rightarrow_{md} \gamma_n$$

*de longitud n, le corresponde una **reducción por la izquierda**:*

$$\gamma_n \Rightarrow_{mi}^{R} \gamma_{n-1} \Rightarrow_{mi}^{R} \dots \Rightarrow_{mi}^{R} S$$

donde en cada paso $\gamma_i \Rightarrow_{mi}^{R} \gamma_{i-1}$ se sustituye la parte derecha de una regla de producción por su parte izquierda, de tal forma que si $\gamma_i = \alpha_1 \beta \alpha_2$ y $\gamma_{i-1} = \alpha_1 A \alpha_2$ entonces $A \to \beta \in P$.

Definición 3.9. *Sea $G = (V_N, V_T, P, E)$ y sea $A \to \beta \in P$.*

En cualquier derivación más a la derecha:

$$S \Rightarrow_{md} \gamma_1 \Rightarrow_{md} \gamma_2 \Rightarrow_{md} \dots \gamma_{n-1} \Rightarrow_{md} \gamma_n$$

*donde $\gamma_{n-1} = \alpha_1 A \alpha_2$ y $\gamma_n = \alpha_1 \beta \alpha_2$, se llama **pivote** de la forma sentencial derecha γ_n a la cadena β, es decir, a la parte derecha de la regla $A \to \beta$.*

Por tanto el pivote de una forma sentencial derecha γ se puede calcular obteniendo, primero, una derivación más a la derecha hasta γ. En la última derivación directa más a la derecha se aplica cierta regla de producción $A \to \beta$. El pivote es, precisamente, β.

Ejemplo 3.2. Consideremos una gramática con el siguiente conjunto de producciones:

$$S \to zABz$$
$$B \to CD$$
$$C \to c$$
$$D \to d$$
$$A \to a$$

Podemos comprobar que $\alpha \equiv zAcdz$ es una forma sentencial derecha porque:

$$S \Rightarrow_{md} zABz \Rightarrow_{md} zACDz \Rightarrow_{md} zACdz \Rightarrow_{md} zA\underline{c}dz$$

De acuerdo con la definición anterior, c es el pivote pues es la frase simple situada más a la izquierda.

Definición 3.10. *Un árbol ordenado y etiquetado D es un **árbol de derivación** para una gramática libre de contexto $G(S) = (V_N, V_T, P, S)$ si:*

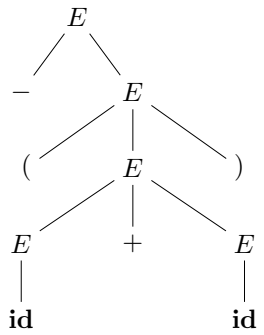
1. *La raíz de D está etiquetada con S.*
2. *Si $D_1, \ldots, D_k$ son los subárboles de los descendientes directos de la raíz, y la raíz de cada D_i está etiquetada con X_i, entonces $S \to X_1 \cdots X_k \in P$. Además D_i debe ser un árbol de derivación en $G(X_i) = (V_N, V_T, P, X_i)$ si $X_i \in V_N$, o bien un nodo hoja con etiqueta X_i si $X_i \in V_T$.*
3. *Alternativamente, si D_1 es el único subárbol de la raíz de D, y la raíz de D_1 tiene como etiqueta λ, entonces $S \to \lambda \in P$.*

En una gramática libre de contexto, una secuencia de derivaciones directas:

$$S \Rightarrow \gamma_1 \Rightarrow \gamma_2 \Rightarrow \ldots \Rightarrow \gamma_{n-1} \Rightarrow \gamma_n \equiv w$$

que conduzcan del símbolo inicial a una sentencia de la gramática puede representarse mediante un *árbol de derivación*. A un mismo árbol pueden corresponderle derivaciones distintas, pero a una derivación le corresponde un único árbol.

Ejemplo 3.3. A las derivaciones más a la izquierda y más a la derecha de la sentencia $-(\mathbf{id} + \mathbf{id})$ mostradas en el ejemplo 3.1 les corresponde el siguiente árbol de derivación:



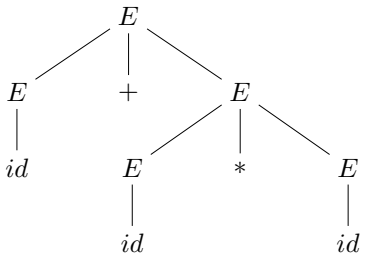
Definición 3.11. *Dada una gramática libre de contexto, se denomina **sentencia ambigua** a una sentencia para la que se pueden encontrar dos o más árboles de derivación distintos, o equivalentemente, dos o más derivaciones más a la izquierda (o más a la derecha) distintas. Se denomina **gramática ambigua** a una gramática que tiene al menos una sentencia ambigua.*

Ejemplo 3.4. La gramática $G = (V_N, V_T, P, E)$ con el siguiente conjunto de reglas de producción:

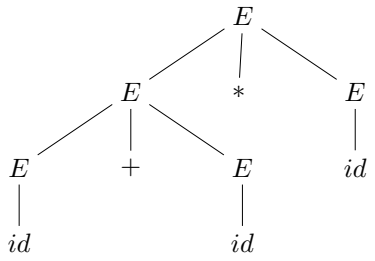
$$E \to E + E$$
$$E \to E * E$$
$$E \to (E)$$
$$E \to id$$

permite representar expresiones aritméticas con operadores $+$ y $*$. El terminal id representa a un identificador de variable cualquiera. Esta gramática es ambigua, puesto que no fija adecuadamente las precedencias y asociatividades de los operadores. Podemos comprobar que existen dos derivaciones más a la derecha distintas de $w \equiv id + id * id$:

$$E \Rightarrow_{md} E + E \Rightarrow_{md} E + E * E \Rightarrow_{md} E + E * \underline{id}$$
$$\Rightarrow_{md} E + id * id \Rightarrow_{md} id + id * id$$



$$E \Rightarrow_{md} E * E \Rightarrow_{md} E * id \Rightarrow_{md} \underline{E + E} * id$$
$$\Rightarrow_{md} E + id * id \Rightarrow_{md} id + id * id$$



Como vemos, la forma sentencial $E + E * id$ tiene dos pivotes posibles: id en el caso de la primera derivación, y $E + E$ en la segunda derivación.

La ambigüedad de una gramática no es deseable, ya que la existencia de múltiples derivaciones de una sentencia puede implicar distintas interpretaciones de la misma y, por tanto, distintas traducciones a código objeto. En el ejemplo anterior, la primera derivación corresponde a la interpretación habitual, ya que se da más precedencia al operador $*$ que al $+$.

Desgraciadamente, no existe un algoritmo que determine si una gramática es ambigua. No obstante, en algunos casos es posible la eliminación de la ambigüedad mediante un estudio del problema que conduzca a la obtención de una gramática equivalente no ambigua. Esto sucederá cuando el lenguaje en sí mismo (no confundir con la gramática) sea no ambiguo. Ahora bien, algunos lenguajes son *inherentemente ambiguos*, es decir, no es posible encontrar una gramática para ellos que no de lugar a sentencias ambiguas.

Afortunadamente, la mayoría de los lenguajes de programación no incluyen un problema de ambigüedad inherente, de modo que se puede resolver la ambigüedad de algunas construcciones habituales (como la precedencia de los operadores aritméticos o la sentencia `if-then-else`) bien con la búsqueda de gramáticas equivalentes, o bien con el uso de soluciones específicas en el diseño del analizador sintáctico.

Analizamos el problema en la siguiente sección.

3.2.2. El problema de la ambigüedad

En la definición 3.11 se presentaron los conceptos de sentencia y gramática ambiguas. En este apartado se realiza un análisis de varios problemas de ambigüedad que pueden aparecer en los lenguajes de programación convencionales, y sus posibles soluciones. Tal y como se indicó en la sección 3.2.1, la ambigüedad de las gramáticas libres de contexto debe resolverse para garantizar que el programa fuente se interpreta correctamente. La dificultad de la ambigüedad estriba en la no existencia de un procedimiento general para su resolución (cuando el lenguaje no es inherentemente ambiguo). Es necesario analizar cada caso y buscar una solución específica que pasa por la transformación de la gramática ambigua, con el fin de obtener otra

equivalente en la que no exista el problema de la ambigüedad[3].

3.2.2.1. Precedencia y asociatividad de operadores

En primer lugar analizamos el problema de la ambigüedad debida a la *precedencia y asociatividad de los operadores*, parcialmente expuesto en el ejemplo 3.4 para el caso de los operadores aritméticos. Problemas equivalentes pueden presentarse para otro tipo de operadores, como los lógicos. Simplificamos la gramática ambigua presentada en el ejemplo, para centrar la discusión en las reglas de producción problemáticas. La gramática que nos interesa analizar, $G = (V_N, V_T, P, E)$, tiene el siguiente conjunto P de reglas de producción:

$$E \rightarrow E + E$$
$$E \rightarrow E * E$$
$$E \rightarrow id$$

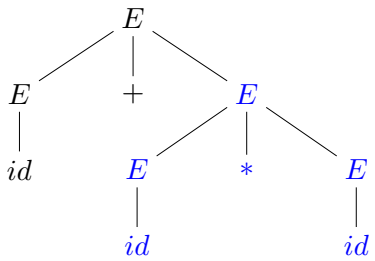
El problema de la ambigüedad de esta gramática se puede dividir en dos partes:

- La ambigüedad debida a la *precedencia* de los operadores.
- La ambigüedad debida a la *asociatividad* de los operadores.

Para ser más precisos, ambos tipos de ambigüedades se deben a que la gramática anterior no expresa adecuadamente la precedencia y la asociatividad de los operadores $+$ y $*$.

Para estudiar el problema relacionado con la *precedencia* de los operadores, debemos observar cómo se derivan, con esta gramática, las expresiones que combinan ambos tipos de operadores. Seleccionamos, por ejemplo, la cadena $w = id + id * id$. Conforme a la definición usual de los operadores $+$ y $*$, el primero tiene menor precedencia que el segundo. Esto supone que la interpretación correcta de la cadena w, usando una notación con paréntesis, es $w = id + (id * id)$.

Como se indicó en el ejemplo 3.4, la gramática G permite hacer dos derivaciones de w, que corresponden a dos árboles de análisis distintos. El árbol que representa la interpretación correcta es el siguiente:



Vemos que el árbol de derivación representa a cada operación en el mismo nivel que sus operandos. El subárbol derecho, marcado en azul, representa a la expresión de multiplicación $id * id$. El resultado de la misma se suma, en el nivel anterior del árbol, con el primer identificador de w. Por tanto, la jerarquía del árbol de derivación es fundamental para la interpretación de las cadenas del lenguaje: las subexpresiones que usan operadores de mayor precedencia deben ser subárboles de aquellas con operadores de menor precedencia.

Observando la gramática G se puede comprobar que el uso de un único no terminal, E, que representa cualquier subexpresión o expresión global (estructura recursiva), no permite distinguir la jerarquía de precedencia de los dos operadores. Por ello cabe plantearse si la incorporación a la gramática de nuevos no terminales podría ayudar a resolver el problema. La respuesta es afirmativa.

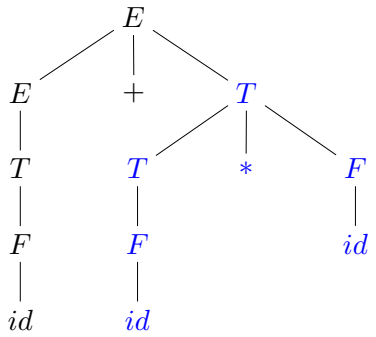
[3] El generador de analizadores sintácticos *Bison* permite resolver la ambigüedad sin modificar la gramática, mediante la definición de precedencias para los tokens conflictivos que permiten eliminar los derivaciones no deseadas.

El no terminal E se puede mantener para representar subexpresiones en las que interviene el operador $+$. Un nuevo no terminal, T (término), se puede emplear para representar subexpresiones con el operador $*$. Finalmente, se puede añadir un tercer no terminal, F (factor), para representar identificadores. Este último no terminal no es estrictamente necesario para eliminar la ambigüedad debida a la precedencia de los operadores, pero contribuye a dotar de una jerarquía clara a toda la gramática. Las reglas de producción de la nueva gramática equivalente son las siguientes:

$$E \to E + T \mid T$$
$$T \to T * F \mid F$$
$$F \to id$$

Cualquier cadena generada por la versión inicial de la gramática G se puede generar con este nuevo conjunto de reglas. Por otra parte, existe una única derivación más a la derecha y, por tanto, un único árbol de derivación de la expresión $w = id + id * id$:

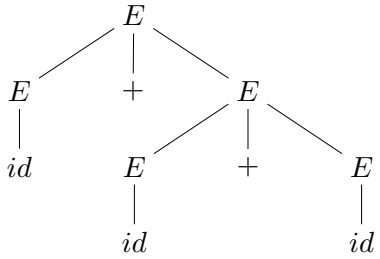
$$E \Rightarrow_{md} E + T \Rightarrow_{md} E + T * F \Rightarrow_{md} E + T * id \Rightarrow_{md} E + F * id \Rightarrow_{md} E + id * id$$
$$\Rightarrow_{md} T + id * id \Rightarrow_{md} F + id * id \Rightarrow_{md} id + id * id$$



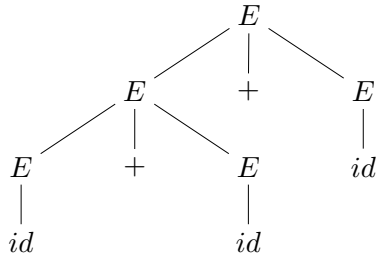
De acuerdo con la estructura de las reglas de producción de la nueva gramática, se puede observar que los no terminales T siempre aparecen como hijos de los nodos correspondientes a los no terminales E, y nunca puede suceder lo contrario. De este modo, queda resuelto el problema de la ambigüedad debida a la precedencia de los operadores.

Volviendo a la versión inicial de las reglas de producción de la gramática G, vemos que también presentan un problema de ambigüedad debido a que no definen la *asociatividad* (izquierda o derecha) de los operadores. Para comprobarlo debemos seleccionar una cadena que use dos o más operadores del mismo tipo. Por ejemplo, estudiamos el caso de $w = id + id + id$. Esta cadena, en la versión inicial de G, permite dos derivaciones más a la derecha, con sus respectivos árboles de derivación:

$$E \Rightarrow_{md} E + E \Rightarrow_{md} E + E + E \Rightarrow_{md} E + E + id$$
$$\Rightarrow_{md} E + id + id \Rightarrow_{md} id + id + id$$



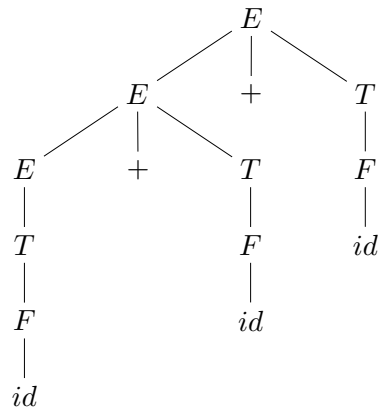
$$E \Rightarrow_{md} E + E \Rightarrow_{md} E + id \Rightarrow_{md} E + E + id$$
$$\Rightarrow_{md} E + id + id \Rightarrow_{md} id + id + id$$



En este caso, la interpretación más frecuentemente implementada es la que da asociatividad izquierda a los operadores binarios aritméticos, de modo que la derivación y el árbol correctos son los mostrados a la derecha, ya que en este caso la cadena se trata como $w = (id + id) + id$.

El problema de la asociatividad no es grave desde el punto de vista de la generación de código, puesto que el operador $+$ es asociativo en su sentido matemático, es decir, $(id + id) + id = id + (id + id)$. No obstante, desde el punto de vista de los métodos de análisis que veremos más adelante, se requiere un tratamiento sistemático que no dé lugar a la ambigüedad. Por tanto, es necesario modificar la gramática para resolver esta situación. Afortunadamente, el mismo conjunto de reglas de producción que resolvían la ambigüedad debida a la precedencia también solucionan la ambigüedad debida a la asociatividad. La única derivación más a la derecha (y el único árbol de derivación) para la cadena $w = id + id + id$, usando el nuevo conjunto de reglas, es el siguiente:

$$E \Rightarrow_{md} E + T \Rightarrow_{md} E + F \Rightarrow_{md} E + id \Rightarrow_{md} E + T + id \Rightarrow_{md} E + F + id$$
$$\Rightarrow_{md} E + id + id \Rightarrow_{md} T + id + id \Rightarrow_{md} F + id + id \Rightarrow_{md} id + id + id$$



La asociatividad izquierda se consigue con las reglas de producción $E \Rightarrow E + T$ y $T \Rightarrow T * F$. Al comenzar el cuerpo de estas reglas con el uso recursivo de los no terminales E y T respectivamente, estas reglas están imponiendo la asociatividad izquierda de los operadores $+$ y $*$.

3.2.2.2. Sentencias `if-then` e `if-then-else`

Otro ejemplo clásico de los problemas de ambigüedad en los lenguajes de programación es el de las gramáticas que incluyen sentencias del tipo `if-then` e `if-then-else` (ambas simultáneamente). Supongamos una gramática que incluye las siguientes reglas de producción:

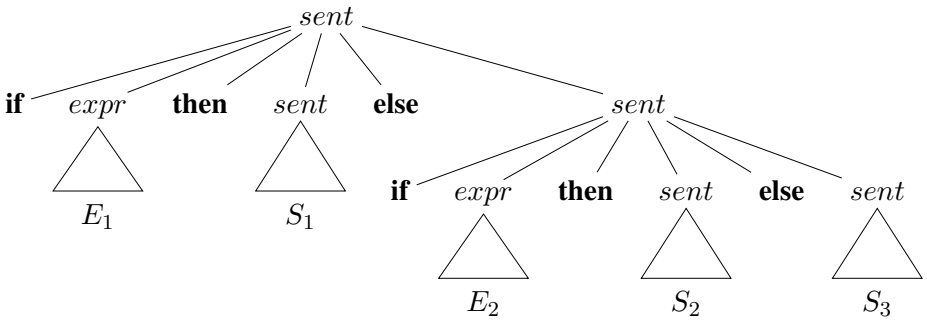
$$\begin{array}{lcl} sent & \rightarrow & \textbf{if } expr \textbf{ then } sent \\ & | & \textbf{if } expr \textbf{ then } sent \textbf{ else } sent \\ & | & S \\ expr & \rightarrow & E \end{array}$$

En esta gramática las palabras clave que aparecen en negrita se tratan como terminales (tokens), mientras que los símbolos $sent$ y $expr$ son no terminales que representan a una sentencia y una expresión respectivamente. Por otra parte, S representa la secuencia de símbolos (tokens y no terminales) de cualquier sentencia que no sea `if-then` ni `if-then-else`, mientras que E representa a la secuencia de símbolos de una expresión cualquiera. Se emplean subíndices en S y E para distinguir diferentes ocurrencias de secuencias y expresiones.

De acuerdo con esta gramática, la siguiente sentencia:

if E_1 **then** S_1 **else if** E_2 **then** S_2 **else** S_3

no es ambigua, ya que tiene un único árbol de derivación:



Sin embargo, la gramática es ambigua porque esta otra sentencia:

if E_1 **then if** E_2 **then** S_1 **else** S_2

tiene dos árboles de derivación, que se muestran en la figura 3.1.

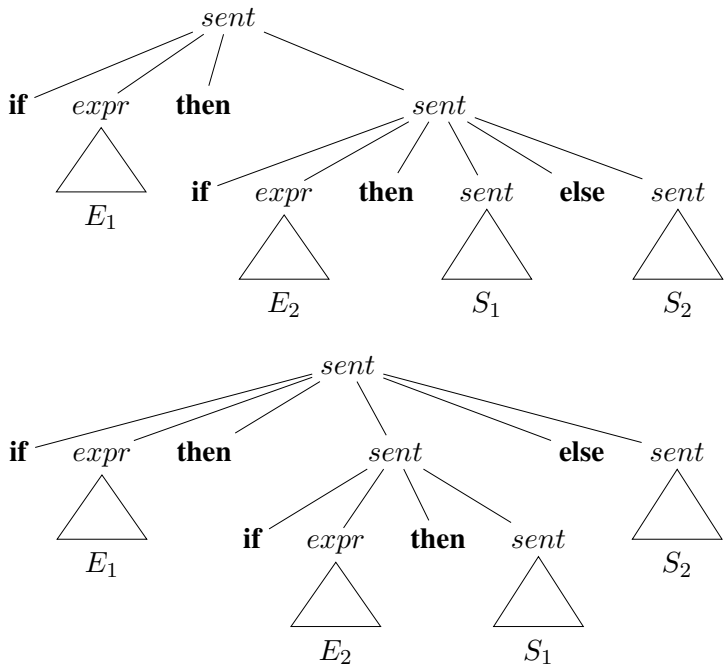


Figura 3.1: Árboles de derivación de una sentencia `if-then-else` ambigua.

En general, el problema de ambigüedad aparece en sentencias en las que hay un **else** que se puede asociar a varios **if** anteriores. En todos los lenguajes de programación con sentencias condicionales como éstas, se prefiere el primer árbol de derivación, que sigue la regla de asociar el **else** al **if** más próximo.

Existen varios enfoques para solucionar la ambigüedad de esta gramática. El primero consiste en transformar la gramática, y la definición del lenguaje, para que las construcciones `if-then-else` tengan *delimitadores de bloque*, de modo que el programador se vea obligado a asociar los **else** con los **if** de forma explícita[4]. Siguiendo esta alternativa, la nueva gramática del lenguaje modificado puede incluir las siguientes reglas:

[4]Es la solución más drástica, ya que influye en el diseño del lenguaje de programación. Lenguajes como Modula o Ada emplean este enfoque.

$$
\begin{array}{rcl}
sent & \rightarrow & \textbf{if } expr \textbf{ then } sent \textbf{ endif} \\
 & | & \textbf{if } expr \textbf{ then } sent \textbf{ else } sent \textbf{ endif} \\
 & | & S \\
expr & \rightarrow & E
\end{array}
$$

de modo que la sentencia ambigua anterior se debe re-escribir de una de las dos siguientes formas (la primera sentencia asocia el **else** al segundo **if**, mientras que la segunda lo asocia al primer **if**):

$$\textbf{if } E_1 \textbf{ then if } E_2 \textbf{ then } S_1 \textbf{ else } S_2 \textbf{ endif endif}$$

$$\textbf{if } E_1 \textbf{ then if } E_2 \textbf{ then } S_1 \textbf{ endif else } S_2 \textbf{ endif}$$

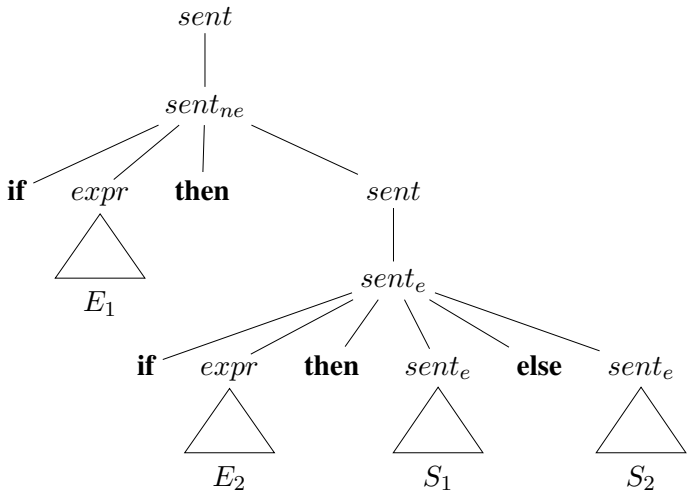
Una segunda alternativa para resolver la ambigüedad de las sentencias `if-then-else` sin modificar el lenguaje consiste en transformar la gramática en otra equivalente que no sea ambigua. Para ello es conveniente distinguir entre las sentencias emparejadas y las sentencias no emparejadas:

- Una *sentencia emparejada* ($sent_e$) es o bien una sentencia `if-then-else` que no contenga sentencias sin emparejar, o bien cualquier otra clase de sentencia no condicional.
- Una *sentencia no emparejada* ($sent_{ne}$) es una sentencia `if-then` o bien una sentencia `if-then-else` que termina con una sentencia no emparejada.

No se permite que las sentencias `if-then-else` tengan una sentencia intermedia no emparejada. De esta forma se evitan árboles como el segundo mostrado en la figura 3.1. La gramática equivalente no ambigua tiene el siguiente conjunto de reglas de producción:

$$
\begin{array}{rcl}
sent & \rightarrow & sent_e \\
 & | & sent_{ne} \\
sent_e & \rightarrow & \textbf{if } expr \textbf{ then } sent_e \textbf{ else } sent_e \\
 & | & S \\
sent_{ne} & \rightarrow & \textbf{if } expr \textbf{ then } sent \\
 & | & \textbf{if } expr \textbf{ then } sent_e \textbf{ else } sent_{ne} \\
expr & \rightarrow & E
\end{array}
$$

La sentencia que era ambigua con la gramática inicial, ahora tiene un único árbol de derivación:



Normalmente no se emplea esta segunda alternativa, ya que tiene únicamente interés teórico. Como veremos más adelante, lo habitual es incluir en el analizador sintáctico un caso especial y ad-hoc para las

construcciones `if-then-else` de modo que se imponga siempre la regla que asocia el **else** al **if** más próximo. Es importante indicar que los compiladores que resuelven la ambigüedad de esta forma, también permiten emplear sentencias que representan bloques de instrucciones, delimitados por marcadores como `{` y `}`. Haciendo uso de estas construcciones, el programador puede crear programas que produzcan árboles de derivación como el segundo indicado en la figura 3.1:

```
1 if (a) {
2       if (b) printf("b\n");
3 }
4 else printf("!a\n");
```

El lector debe comprender, en este punto, la diferencia entre el anterior bloque de código C y el siguiente, que no emplea `{`, `}`:

```
1 if (a)
2       if (b) printf("b\n"); else printf("!a\n");
```

3.2.3. Autómatas de pila

En el tema anterior se emplearon los autómatas finitos para el reconocimiento de los lenguajes descritos con expresiones regulares. En este tema se hace uso de los *autómatas de pila* para reconocer los lenguajes descritos con las gramáticas libres de contexto.

La diferencia fundamental entre los autómatas finitos y los autómatas de pila es que estos últimos disponen de una memoria con estructura de pila que modifican al realizar sus transiciones. Esta memoria permite la construcción de una derivación para la cadena de tokens de la entrada. Sin embargo, ambos tipos de autómata comparten el principio básico de que la transición entre estados depende del símbolo actual de la entrada[5] y del estado actual del autómata.

Definición 3.12. *Un autómata de pila es una 7-tupla* $M \equiv (Q, V, \Sigma, \delta, q_0, z_0, F)$ *donde:*

- Q *es un conjunto finito de estados.*
- V *es el alfabeto de entrada.*
- Σ *es el alfabeto de la pila.*
- q_0 *es el estado inicial.*
- z_0 *es el símbolo inicial de la pila.*
- $F \subseteq Q$ *es el conjunto de estados finales.*
- δ *es la función de transición, definida del siguiente modo:*

$$\delta \ : \ Q \times (V \cup \{\lambda\}) \times \Sigma \rightarrow P(Q \times \Sigma^*)$$

Cuando se emplea un autómata de pila para el reconocimiento del lenguaje definido por una gramática libre de contexto, el alfabeto de entrada V del autómata coincide con el conjunto de tokens del lenguaje, es decir, el conjunto de símbolos terminales V_T. Por otra parte, el alfabeto de la pila Σ contiene a los conjuntos de símbolos terminales V_T y no terminales V_N del lenguaje. También se emplea el símbolo inicial de la gramática S como símbolo inicial de la pila z_0.

[5] Para el analizador léxico el símbolo es un carácter del programa fuente. Para el analizador sintáctico el símbolo es un token.

Definición 3.13. *Se denomina **configuración de un autómata de pila** a su situación en un instante determinado, expresada formalmente por medio de una tripla $(q, w, \alpha) \in (Q \times V^* \times \Sigma^*)$, donde:*

- *$q \in Q$ es el estado actual del autómata.*
- *$w \in V^*$ es la subcadena de entrada que aún no se ha analizado. El siguiente símbolo de la entrada se representa en el extremo izquierdo de la cadena w.*
- *$\alpha \in \Sigma^*$ es el contenido actual de la pila. La cima de la pila se representa en el extremo izquierdo de la cadena α.*

Con $w = \lambda$ se representa que se ha consumido la entrada completamente. Y con $\alpha = \lambda$ se representa a la pila vacía.

Definición 3.14. *Un movimiento de un autómata de pila es una transición entre dos configuraciones:*

$$(q_i, aw, z\alpha) \Rightarrow (q_j, w, \beta\alpha)$$

con $q_i, q_j \in Q$, $a \in (V \cup \{\lambda\})$, $z \in \Sigma$, $w \in V^*$ y $\alpha, \beta \in \Sigma^*$. El movimiento es válido siempre y cuando:

$$(q_j, \beta) \in \delta(q_i, a, z)$$

Como podemos ver, el movimiento emplea el estado actual q_i como primer argumento de la función de transición, el siguiente símbolo de la entrada a como segundo argumento, y el símbolo de la cima de la pila z como tercer argumento. Es importante señalar que el autómata no puede realizar ningún movimiento si la pila está vacía. También es interesante observar que, cuando $a = \lambda$, se produce un movimiento en el que no se consume ningún símbolo de la entrada.

Por la definición de la función de transición, un autómata de pila es *no determinista* ya que, dado un estado, un símbolo del alfabeto de entrada o λ, y un símbolo del alfabeto de la pila, el autómata puede hacer movimientos a varias configuraciones distintas, con distintos estados q_i, consumiendo o no el símbolo actual de la entrada a, y reemplazando el tope de la pila por distintas γ_i:

$$\delta(q, a, z) = \{(q_1, \gamma_1), (q_2, \gamma_2), \ldots, (q_m, \gamma_m)\}$$

La cadena de entrada w reconocida por el autómata será aceptada en dos situaciones:

- Por *estado final* si, partiendo de su configuración inicial (q_0, w, z_0), se llega a una configuración final (q_f, λ, α) empleando movimientos válidos. La cadena de movimientos entre ambas configuraciones se representa con:

  $$(q_0, w, z_0) \Rightarrow^* (q_f, \lambda, \gamma)$$

  donde $q_f \in F$ y $\gamma \in \Sigma^*$.

- Por *vaciado de pila* si, después de consumir toda la cadena de entrada, se llega a una configuración con la pila vacía, independientemente del tipo de estado $q \in Q$ de la misma:

  $$(q_0, w, z_0) \Rightarrow^* (q, \lambda, \lambda)$$

  En este caso, le definición del autómata de pila debe indicar $F = \varnothing$.

Ejemplo 3.5. La gramática $G = (V_N, V_T, P, S)$ con el siguiente conjunto de reglas de producción:

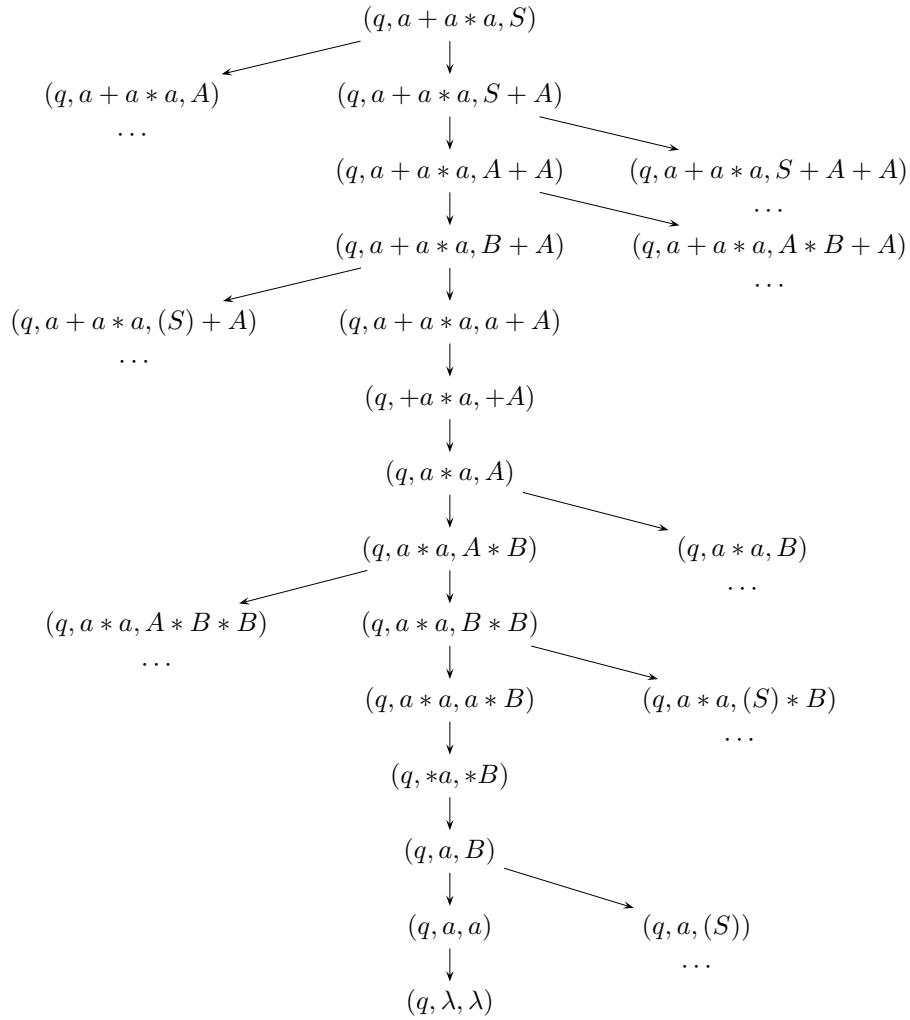
$$
\begin{aligned}
S &\to S + A \\
S &\to A \\
A &\to A * B \\
A &\to B \\
B &\to (S) \\
B &\to a
\end{aligned}
$$

permite representar expresiones aritméticas con operadores $+$ y $*$. En esta gramática, $V_T = \{a, +, *, (,)\}$. El terminal a representa a un identificador de variable cualquiera. Por otra parte, $V_N = \{S, A, B\}$. Podemos definir un automata de pila $M \equiv (Q, V, \Sigma, \delta, q_0, z_0, F)$ para reconocer el lenguaje definido por esta gramática:

- $Q = \{q\}$, es decir, sólo hay un estado.
- $V = V_T$.
- $\Sigma = V_N \cup V_T$.
- $q_0 = q$.
- $z_0 = S$.
- $F = \emptyset$, es decir, no hay estados finales, porque el lenguaje se reconoce por vaciado de pila.
- δ está definida del siguiente modo:

$$
\begin{aligned}
\delta(q, \lambda, S) &= \{(q, S + A), (q, A)\} \\
\delta(q, \lambda, A) &= \{(q, A * B), (q, B)\} \\
\delta(q, \lambda, B) &= \{(q, (S)), (q, a)\} \\
\delta(q, x, x) &= \{(q, \lambda)\}\ \forall x \in V_T
\end{aligned}
$$

Los cálculos para el reconocimiento de $w = a + a * a$ se representan, parcialmente, en el árbol siguiente:



La última configuración corresponde a la situación de vaciado de pila, y por tanto la cadena $w = a + a * a$ se reconoce correctamente.

3.2.4. Características no libres de contexto de los lenguajes de programación

Algunas construcciones típicas de los lenguajes de programación no pueden ser expresadas mediante gramáticas libres de contexto. En este apartado consideramos dos casos representativos, usando abstracciones de los mismos mediante lenguajes simples.

En primer lugar consideramos el caso de la declaración de identificadores previa a su uso en un programa. El lenguaje para modelar esta construcción consiste en cadenas de la forma wcw en donde la primera w representa la declaración de un identificador, c representa un fragmento de programa cualquiera, y la segunda w representa el uso del identificador. El lenguaje abstracto se puede formalizar del siguiente modo:

$$L_1 = \{wcw \mid w \in (\mathbf{a}|\mathbf{b})^*\}$$

L_1 está formado por palabras compuestas por una cadena cualquiera de símbolos a y b separados por una c, como $aabcaab$. El lenguaje L_1 sólo puede ser descrito mediante gramáticas de tipo 1 (sensibles al contexto), lo que implica que los lenguajes de programación C o $Java$, que requieren identificadores de longitud arbitraria que deben estar declarados antes de su uso, tampoco pueden ser descritos únicamente con gramáticas libres de contexto.

Por razones prácticas, los compiladores de los lenguajes de programación representan a todos los identificadores con un mismo token en la gramática, y dejan a la fase de análisis semántico la verificación de que han sido declarados previamente a su uso.

Algo similar sucede con la declaración y el uso de las funciones. En este caso el compilador debe verificar que el número de argumentos empleado en una llamada a una función coincide con el número de parámetros formales declarados en la definición de la función. El lenguaje mediante el que se puede modelar esta situación consiste en cadenas de la forma $a^n b^m c^n d^m$. En este lenguaje, a^n y b^m pueden representar a la lista de parámetros formales de dos funciones que tienen n y m argumentos en su definición, mientras que c^n y d^m representan a la lista de parámetros usados en llamadas a estas dos funciones. Este lenguaje se puede formalizar como se indica a continuación:

$$L_2 = \{a^n b^m c^n d^m \mid n \geq 1 \wedge m \geq 1\}$$

De nuevo, el lenguaje L_2 no es libre de contexto, ya que consiste en cadenas generadas por la expresión regular $\mathbf{a}^*\mathbf{b}^*\mathbf{c}^*\mathbf{d}^*$ tales que el número de a es igual al número de c, y el número de b es igual al número de d. Por ello, la verificación de que el número de parámetros de una llamada es correcto se realiza durante la fase de análisis semántico.

3.3. Clasificación de los métodos de análisis sintáctico

Definición 3.15. *Un **método de análisis** es un procedimiento sistemático cuyo objetivo es, dada una cadena perteneciente al lenguaje generado por una gramática libre de contexto, obtener el árbol de derivación de dicha cadena mediante la aplicación de uno o más algoritmos.*

Para la construcción de la fase de análisis sintáctico de un compilador se suele seleccionar alguno de los métodos de análisis tratado en la bibliografía de la materia, que son el resultado de una sólida teoría. Estos métodos se pueden clasificar en los dos grandes grupos siguientes:

- **Análisis sintáctico descendente**: esta categoría está formada por los métodos que construyen el árbol de derivación a partir de la raíz, etiquetada con el símbolo inicial de la gramática, hasta llegar a las hojas del árbol, etiquetadas con los símbolos terminales o tokens que forman la cadena analizada. Los métodos más interesantes realizan la construcción del árbol de derivación empleando derivaciones izquierdas.
- **Análisis sintáctico ascendente**: los métodos de esta categoría construyen el árbol de derivación a partir de las hojas del árbol, etiquetadas con los símbolos terminales o tokens, hasta llegar a la raíz, etiquetada con el símbolo inicial de la gramática. Los métodos más eficientes emplean reducciones izquierdas (inversas de las derivaciones derechas) para guiar la construcción del árbol de derivación.

Entre los métodos ascendentes y descendentes existen algunos que son **generales**, es decir, que pueden analizar cualquier gramática libre de contexto. Sin embargo, estos métodos son bastante ineficientes.

Afortunadamente, se puede emplear un subconjunto de las gramáticas libres de contexto que son suficientemente expresivas para describir a la mayoría de los lenguajes de programación, y a las que se pueden aplicar métodos de análisis muy eficientes. Estos métodos se estudian en las secciones 3.5 y 3.7.

3.4. Conjuntos PRIMERO y SIGUIENTE

Tanto para los métodos ascendentes como para los descendentes, que veremos más adelante, es necesario definir dos conjuntos de símbolos que son fundamentales para el análisis predictivo: los conjuntos PRIMERO y SIGUIENTE.

3.4.1. Conjunto PRIMERO

Definición 3.16. *Dada una gramática libre de contexto $G = (V_N, V_T, S, P)$ y una cadena de símbolos $\alpha \in (V_N \cup V_T)^*$, se define el conjunto* PRIMERO *de dicha cadena de la siguiente forma:*

$$\text{PRIMERO}(\alpha) = \{a \in V_T \mid \alpha \Rightarrow^* a\beta\} \cup \{\ \textit{if}\ (\alpha \Rightarrow^* \lambda)\ \textit{then}\ \{\lambda\}\ \textit{else}\ \emptyset\}$$

Es decir, el conjunto PRIMERO(α) está formado por todos los terminales que pueden aparecer al comienzo de las cadenas de terminales que se pueden derivar de α. Si α puede derivar en la cadena vacía λ, entonces λ también forma parte del conjunto PRIMERO(α).

Teniendo en cuenta que $\alpha = X_1X_2 \dots X_n$, podemos plantear un algoritmo para la obtención del conjunto PRIMERO(α) que se base en el cálculo de PRIMERO(X), siendo $X \in (V_N \cup V_T)$.

Algoritmo 3.1. Cálculo de PRIMERO(X).
Entrada: una gramática libre de contexto $G = (V_N, V_T, S, P)$, y un símbolo $X \in (V_N \cup V_T)$.
Salida: PRIMERO(X).

Si $X \in V_T$ entonces PRIMERO(X) es $\{X\}$;
en otro caso:

1. Si $X \to \lambda$ entonces añadir λ a PRIMERO(X);
2. Por cada regla $X \to Y_1Y_2 \dots Y_k$:
   2.1. Si para alguna i entre 1 y k se cumple que $Y_1Y_2 \dots Y_{i-1} \Rightarrow^* \lambda$ entonces añadir a PRIMERO(X) todo lo que pertenezca a PRIMERO(Y_i) salvo λ;
   2.2. Si $Y_1Y_2 \dots Y_k \Rightarrow^* \lambda$, es decir, toda la parte derecha de la regla puede derivar en λ, entonces añadir λ a PRIMERO(X);

El algoritmo anterior plantea la construcción del conjunto PRIMERO de un símbolo X de la gramática en función de los conjuntos PRIMERO de otros símbolos Y_i de la gramática.

Es necesario realizar la construcción de todos estos conjuntos simultáneamente, y en el caso de que exista alguna dependencia circular entre varios conjuntos PRIMERO, el algoritmo finaliza cuando no se puedan añadir más elementos a ningún conjunto siguiendo las reglas indicadas.

Apoyándose en el algoritmo anterior, es posible plantear un algoritmo para el cálculo de PRIMERO(α), siendo $\alpha = X_1X_2 \dots X_n$.

Algoritmo 3.2. Cálculo de PRIMERO($X_1X_2 \dots X_n$).
Entrada: una gramática libre de contexto $G = (V_N, V_T, S, P)$, y una cadena $X_1X_2 \dots X_n \in (V_N \cup V_T)^*$.
Salida: PRIMERO($X_1X_2 \dots X_n$).

---

1. REPETIR desde $i = 1$ HASTA n:
   1.1. Añadir a PRIMERO($X_1X_2 \dots X_n$) todo lo que pertenezca a PRIMERO(X_i) salvo λ;
   1.2. $X_1 \dots X_i \not\Rightarrow^* \lambda$ TERMINAR;
2. Si $X_1X_2 \dots X_n \Rightarrow^* \lambda$ añadir λ a PRIMERO($X_1X_2 \dots X_n$);

El algoritmo de cálculo de PRIMERO($X_1X_2 \dots X_n$) comienza añadiendo a este conjunto todos los símbolos distintos de λ de PRIMERO(X_1). Si λ está en PRIMERO(X_1), continúa añadiendo los símbolos distintos de λ de PRIMERO(X_2), y así sucesivamente. Por último, se añade λ a PRIMERO($X_1X_2 \dots X_n$) en el caso de que λ pertenezca a PRIMERO(X_i) para todo i.

3.4.2. Conjunto SIGUIENTE

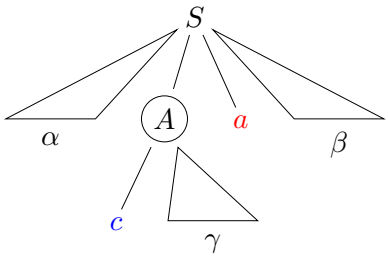
Definición 3.17. *Dada una gramática libre de contexto $G = (V_N, V_T, S, P)$ y un no terminal $A \in V_N$, se define el conjunto* SIGUIENTE *de dicho no terminal de la siguiente forma:*

$$\text{SIGUIENTE}(A) = \{a \in V_T \mid S \Rightarrow^+ \alpha A a \beta\} \cup \{\ \textit{if}\ (S \Rightarrow^* \alpha A)\ \textit{then}\ \{\$\}\ \textit{else}\ \emptyset\}$$

Es decir, el conjunto SIGUIENTE se define como el conjunto de símbolos terminales a que pueden aparecer inmediatamente a la derecha de A en alguna forma sentencial de la gramática.

Si A es el símbolo más a la derecha en alguna forma sentencial de la gramática, entonces el símbolo especial \$, que representa el final de la entrada, forma también parte del conjunto SIGUIENTE(A).

Se pueden expresar gráficamente los conjuntos PRIMERO(A) y SIGUIENTE(A) con el siguiente árbol:



Con el fin de simplificar la representación, se muestra en un único nivel del árbol la cadena de derivaciones $S \Rightarrow^+ \alpha A a \beta$.

Los terminales pertenecientes a SIGUIENTE(A) están representados por los símbolos $a \in V_T$ que pueden aparecer a continuación de A en una forma sentencial.

Por otro lado, los terminales pertenecientes a PRIMERO(A) serán aquellos que, como $c \in V_T$ en el árbol de ejemplo, pueden aparecer al comienzo de las cadenas derivadas desde A.

Algoritmo 3.3. Cálculo de SIGUIENTE(A).
Entrada: una gramática libre de contexto $G = (V_N, V_T, S, P)$, y un símbolo $A \in V_N$.
Salida: SIGUIENTE(A).

1. Si A es el símbolo inicial de la gramática S, añadir \$ a SIGUIENTE(A);
   El símbolo \$ representa el delimitador derecho de la entrada.
2. Por cada producción de la forma $B \to \alpha A \beta$:
   Añadir a SIGUIENTE(A) todo lo que esté en PRIMERO(β) salvo λ;
3. Por cada producción de la forma $B \to \alpha A$ o de la forma $B \to \alpha A \beta$ donde $\lambda \in$ PRIMERO(β):
   Añadir a SIGUIENTE(A) todo lo que esté en SIGUIENTE(B);

Al igual que en el caso del algoritmo de construcción del conjunto PRIMERO, el conjunto SIGUIENTE de un símbolo A de la gramática puede obtenerse en función de los conjuntos PRIMERO y SIGUIENTE de otros símbolos X de la gramática. Por tanto, antes de aplicar este algoritmo de cálculo de los conjuntos SIGUIENTE, es necesario calcular los conjuntos PRIMERO, y a continuación se debe realizar la construcción de todos los conjuntos SIGUIENTE simultáneamente hasta que no se puedan añadir más elementos a ningún conjunto siguiendo las reglas indicadas.

Ejemplo 3.6. Sea la siguiente gramática:

$$\begin{array}{lcl} A & \rightarrow & B\,e \mid a \\ B & \rightarrow & C\,D \mid b \\ C & \rightarrow & c \mid \lambda \\ D & \rightarrow & d \mid \lambda \end{array}$$

Obtenemos a continuación los conjuntos PRIMERO de todos los símbolos, empleando el algoritmo 3.1:

- Los conjuntos PRIMERO para todos los símbolos terminales $V_T = \{a, b, c, d, e\}$ son ellos mismos.
- Por la regla de producción $A \rightarrow a$ se añade a a PRIMERO(A).
- Por la regla de producción $B \rightarrow b$ se añade b a PRIMERO(B).
- Por la regla de producción $C \rightarrow c$ se añade c a PRIMERO(C).
- Por la regla de producción $C \rightarrow \lambda$ se añade λ a PRIMERO(C).
- Por la regla de producción $D \rightarrow d$ se añade d a PRIMERO(D).
- Por la regla de producción $D \rightarrow \lambda$ se añade λ a PRIMERO(D).

Hasta este punto, tenemos los siguientes conjuntos PRIMERO:

$$\begin{array}{l} \text{PRIMERO}(A) = \{\ a,\ \cdots\ \} \\ \text{PRIMERO}(B) = \{\ b,\ \cdots\ \} \\ \text{PRIMERO}(C) = \{\ c,\ \lambda\ \} \\ \text{PRIMERO}(D) = \{\ d,\ \lambda\ \} \end{array}$$

Pasamos a analizar la regla $B \rightarrow CD$:

- Como C aparece al comienzo del lado derecho de la regla, tenemos que añadir a PRIMERO(B) todo lo que contenga PRIMERO(C), salvo λ. Por tanto, añadimos c a PRIMERO(B).
- Teniendo en cuenta que PRIMERO(C) contiene λ, hay que añadir a PRIMERO(B) los símbolos que contiene PRIMERO(D) salvo λ, es decir, d.
- Como tanto PRIMERO(C) como PRIMERO(D) contienen λ, hay que añadir también λ a PRIMERO(B).

Con lo anterior, podemos actualizar los conjuntos PRIMERO:

$$\text{PRIMERO}(B) = \{\ b,\ c,\ d,\ \lambda\ \}$$

Analizamos ahora la regla $A \rightarrow Be$:

- Como B aparece al comienzo del lado derecho de la regla, tenemos que añadir a PRIMERO(A) todo lo que contenga PRIMERO(B), salvo λ. Por tanto, añadimos b, c y d a PRIMERO(A).
- Teniendo en cuenta que PRIMERO(B) contiene λ, hay que añadir a PRIMERO(A) el símbolo e.

La versión final de los conjuntos PRIMERO incluirá:

$$\text{PRIMERO}(A) = \{\ a,\ b,\ c,\ d,\ e\ \}$$

Pasamos ahora a calcular los conjuntos SIGUIENTE:

- Para el no terminal A, el conjunto SIGUIENTE(A) incluye el símbolo \$, por ser el símbolo inicial.
- Para el no terminal B, teniendo en cuenta que aparece seguido de e en la regla $A \to Be$, añadimos e al conjunto SIGUIENTE(B).
- Por la regla $B \to CD$, debemos añadir a SIGUIENTE(C) todo lo que contiene PRIMERO(D) salvo λ, es decir, añadimos d. Al contener λ el conjunto PRIMERO(D), añadimos también a SIGUIENTE(C) los símbolos de SIGUIENTE(B), es decir, e.
- Por la regla $B \to CD$, al encontrarse D en el extremo derecho del cuerpo de la regla, añadimos a SIGUIENTE(D) los símbolos de SIGUIENTE(B), es decir, e.

Los conjuntos SIGUIENTE obtenidos son los siguientes:

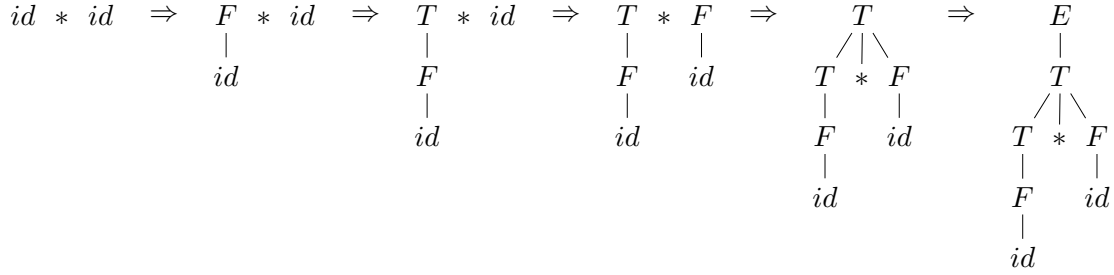
$$\begin{aligned}\text{SIGUIENTE}(A) &= \{\ \$\ \}\\ \text{SIGUIENTE}(B) &= \{\ e\ \}\\ \text{SIGUIENTE}(C) &= \{\ d,\ e\ \}\\ \text{SIGUIENTE}(D) &= \{\ e\ \}\end{aligned}$$

3.5. Análisis ascendente

Los analizadores ascendentes construyen el árbol de análisis de forma inversa, es decir, comenzando por la cadena de entrada, situada en los nodos hoja, y generando los nodos intermedios hasta llegar a la raiz del árbol, etiquetada con el símbolo inicial de la gramática. Consideremos la siguiente gramática, muy parecida a la empleada en el apartado 3.2.2.1:

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

La siguiente secuencia de árboles ilustra el análisis de la cadena de tokens de entrada $id * id$:



En esta sección se presenta un método de análisis ascendente sin retroceso que opera empleando una técnica conocida como *reducción-desplazamiento* (shift-reduce). Esta denominación hace referencia a las dos acciones básicas recogidas en la tabla de análisis:

- Una acción de *reducción* consiste en la aplicación de modo inverso de una regla de producción de la gramática a la forma sentencial actual.
- Una acción de *desplazamiento* consiste en el paso del símbolo actual de la entrada a la cima de la pila.

La decisión clave durante el análisis ascendente consiste en decidir cuándo desplazar o reducir, y en este último caso, qué regla de producción aplicar.

Ejemplo 3.7. En la secuencia de árboles anterior, la forma sentencial actual viene determinada por los nodos en la base del árbol actual. Por tanto, la secuencia de formas sentenciales de este análisis es:

$$id * id,\ F * id,\ T * id,\ T * F,\ T,\ E$$

La secuencia comienza con la cadena de entrada. La primera reducción reduce el símbolo id más a la izquierda a F, usando la regla de producción $F \rightarrow id$. Opcionalmente se podría haber reducido el símbolo id más a la derecha. Sin embargo, el método que estudiaremos opera aplicando *reducciones más a la izquierda*, es decir, reduce primero los símbolos situados en el lado izquierdo de la forma sentencial actual.

La segunda reducción reduce F a T, obteniendo $T * id$. Obsérvese que, en este momento, existe la posibilidad de reducir T a E empleando la regla $E \rightarrow T$. Sin embargo, esta reducción conduciría a la secuencia de símbolos $E * id$ que no es una forma sentencial, es decir, que no se puede derivar desde el símbolo inicial de la gramática. Por ello, se escoge la reducción de id a F, resultando $T * F$. A continuación se reduce $T * F$ a T, y finalmente el análisis se completa reduciendo T a E.

El analizador ascendente construye de forma inversa una derivación. En concreto, la secuencia de árboles anterior conduce a la derivación:

$$E \Rightarrow T \Rightarrow T * F \Rightarrow T * id \Rightarrow F * id \Rightarrow id * id$$

Como se puede observar, se trata de una derivación más a la derecha (véase la definición 3.8). Por tanto, a una reducción más a la izquierda le corresponde una derivación más a la derecha.

Se puede construir un analizador ascendente que haga exactamente lo contrario, es decir, que emplee reducciones más a la derecha para construir una derivación más a la izquierda. Sin embargo, este tipo de analizador es menos práctico. La razón estriba en que sería necesario recorrer toda la cadena de entrada para llegar al final y aplicar la primera reducción. En el caso del procesamiento de grandes programas, esto exigiría una gran cantidad de memoria. Por el contrario, un analizador que emplea reducciones más a la izquierda puede comenzar a aplicarlas tras la lectura de unos pocos símbolos de la entrada.

3.5.1. Gramáticas LR

El análisis ascendente predictivo no se puede aplicar a cualquier gramática libre de contexto, sino sólo a un subconjunto que permite decidir de forma determinista la acción a realizar (reducir o desplazar) y en caso de reducir, la producción a aplicar. A este subconjunto se le denomina gramáticas $LR(k)$. Las siglas significan:

- **L**: la entrada se examina de izquierda a derecha (left to right).
- **R**: se construye la derivación más a la derecha (rightmost derivation) en orden inverso.
- **k**: se necesita examinar k tokens de anticipación para decidir qué acción realizar en cada paso.

En esta asignatura estudiaremos las gramáticas $LR(1)$, ya que son suficientemente potentes como para tratar la mayoría de las construcciones de los lenguajes de programación.

La principal desventaja del método de análisis ascendente predictivo basado en las gramáticas LR es la cantidad de trabajo que supone la construcción a mano de un reconocedor para un lenguaje de programación normal. Afortunadamente, existen herramientas que generan de forma automática estos reconocedores, partiendo de una especificación de la gramática del lenguaje.

3.5.2. Fundamentos del análisis LR predictivo

La técnica de análisis predictivo por *reducción-desplazamiento* se apoya en un autómata de pila determinista que realiza el análisis de una cadena w de forma ascendente, empleando reducciones más a la izquierda:

$$w \equiv \gamma_n \Rightarrow^R_{mi} \gamma_{n-1} \Rightarrow^R_{mi} \ldots \Rightarrow^R_{mi} \gamma_1 \Rightarrow^R_{mi} \gamma_0 \equiv S$$

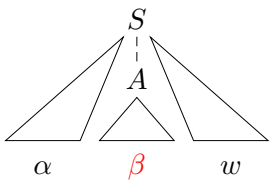
Para reducir desde γ_i a γ_{i-1}, el autómata busca en la pila el pivote (véase la definición 3.9) de la forma sentencial derecha actual γ_i. Supongamos que $\gamma_i = \alpha\beta z$, y que el pivote es β. El analizador debe sustituir β

por el lado izquierdo de la regla de producción escogida para reducir. Supongamos que la regla es $A \to \beta$. En este caso, la nueva forma sentencial derecha es $\gamma_{i-1} = \alpha A z$.

Para realizar esta operación, el analizador emplea una pila en la que mantiene una parte de la forma sentencial actual. En concreto, antes de reducir el analizador encuentra la cadena $\alpha\beta$ en la pila, estando β en la cima, mientras que $z \in V_T^*$ son los símbolos pendientes de ser tratados en la entrada. El primero de estos símbolos puede ser inspeccionado como símbolo de anticipación para realizar el análisis $LR(1)$, ayudando a la elección de la regla de producción con la que se realiza la reducción.

Para llegar a disponer de la cadena $\alpha\beta$ con el pivote de la forma sentencial actual en la cima de la pila, el analizador puede realizar previamente desplazamientos de símbolos desde la entrada a la pila, avanzando el puntero de lectura de la entrada.

La siguiente figura muestra la acción de reducción del pivote β al no terminal del lado izquierdo de la regla de producción:



Esta acción equivale a crear un nuevo nodo en la construcción ascendente del árbol de análisis. La localización del pivote se realiza con ayuda de una tabla de análisis, cuya construcción se puede llevar a cabo empleando tres métodos distintos, que se estudian en secciones posteriores:

- *Método Simple-LR (SLR)*: es el más sencillo y menos potente.
- *Método LR-canónico*: es el más costoso y también el más potente.
- *Método Lookahead-LR ($LALR$)*: es un método que llega a un compromiso entre los dos anteriores.

Cuando un analizador LR emplea una tabla de análisis SLR, LR-canónica o $LALR$, el analizador recibe el nombre de la tabla empleada y la gramática que genera el lenguaje reconocido por el analizador también se denonima con el método usado para construir la tabla.

Existe la posibilidad de que se produzcan conflictos (varias acciones posibles en una misma configuración) durante la construcción de la tabla de análisis con cualquiera de los métodos empleados. Cuando sucede esto, se dice que la gramática no es del tipo correspondiente (SLR, LR-canónica o $LALR$), y es necesario o bien transformar la gramática para llegar a otra equivalente que no genere conflictos o bien seleccionar de forma manual la acción adecuada de acuerdo con un criterio de eliminación de los conflictos que esté justificado.

Pueden aparecer dos tipos de conflicto durante la construcción de una tabla de análisis ascendente predictivo:

- *Conflicto desplaza/reduce*: la configuración actual permite tanto el desplazamiento del símbolo actual de la entrada a la cima de la pila como la reducción empleando una regla de producción.
- *Conflicto reduce/reduce*: la configuración actual permite emplear varias reglas de producción para realizar una reducción.

En próximas secciones veremos ejemplos de ambos tipos de conflicto, así como posibles modos de resolverlos.

3.5.3. Algoritmo de análisis sintáctico ascendente predictivo

Independientemente del método que empleemos para construir la tabla, el análisis LR siempre emplea el mismo algoritmo. Para entender su funcionamiento, es necesario realizar varias definiciones previas.

Definición 3.18. *Se denomina* configuración del analizador LR *a un par:*

$$(s_0\ X_1\ s_1\ X_2\ s_2 \ldots X_m\ s_m\ ,\ a_i\ a_{i+1}\ \ldots\ a_n\ \$)$$

donde $X_i \in \{V_N \cup V_T\}$, $a_i \in V_T$ *y* s_i *son estados del autómata de pila. El primer elemento del par representa el contenido de la* pila *del analizador, con el tope en el extremo derecho, mientras que el segundo elemento representa la cadena de símbolos de entrada que queda por reconocer. La configuración del analizador permite deducir la forma sentencial actual de la cadena de reducciones:*

$$X_1 X_2 \ldots X_m a_i a_{i+1} \ldots a_n$$

Por conveniencia para el estudio del algoritmo de análisis LR, en esta asignatura adoptamos la convención de indicar en el primer elemento de la configuración del analizador los símbolos X_i. No obstante, esta indicación es prescindible, ya que la secuencia de estados $s_0 \ldots s_i$ permite (como se indicará seguidamente) deducir la secuencia de símbolos $X_1 \ldots X_i$. Cada estado de la secuencia corresponde a un símbolo salvo el estado inicial s_0, que actúa como un marcador del fondo de la pila.

La figura 3.2 muestra el diseño básico del analizador LR.

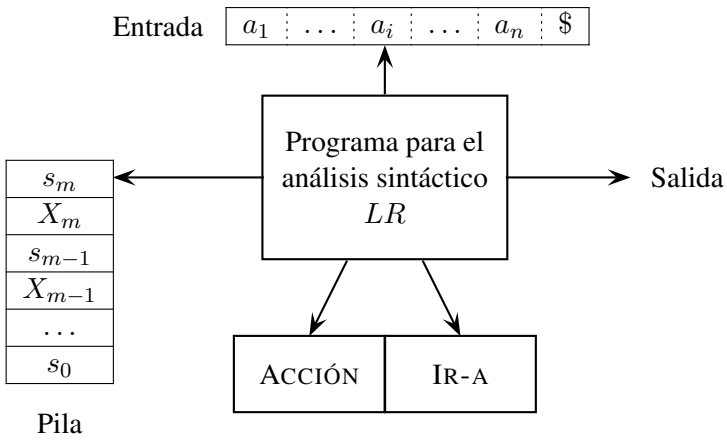


Figura 3.2: Modelo de un analizador LR.

La tabla de análisis de un analizador LR consta de dos partes: ACCIÓN e IR-A.

La tabla ACCIÓN es bi-dimensional, y se indexa con un estado i y un símbolo $a \in V_T \cup \{\$\}$. La tabla se emplea usando como estado el indicado en la cima de la pila y como símbolo el referenciado por el apuntador de la entrada. El contenido de la entrada ACCIÓN$[i, a]$ puede ser:

- *Desplaza* j, donde j es un estado. La acción realizada por el parser consiste en apilar el símbolo a seguido del estado j.
- *Reduce* $A \to \beta$. La acción realizada por el analizador consiste en desapilar tantos símbolos y estados como indique la longitud de la cadena β, para apilar a continuación el símbolo A.
- *Aceptar*. El analizador acepta la entrada y finaliza el análisis.
- *Error*. El analizador informa del error en la entrada y toma alguna medida para continuar el análisis.

La tabla IR-A también es bidimensional, y se indexa con un estado i y un símbolo $A \in V_N$. Nuevamente, la tabla se emplea usando el estado indicado en la cima de la pila, mientras que el símbolo A corresponde a la parte izquierda de una regla de producción que se acaba de emplear para reducir. El valor de la entrada IR-A$[i, A]$ es el estado j al que pasa el analizador, que es inmediatamente apilado a continuación de A.

Sea la configuración actual del analizador $(s_0\ X_1\ s_1\ X_2\ s_2 \ldots X_m\ s_m\ ,\ a_i\ a_{i+1}\ \ldots\ a_n\ \$)$; las acciones de desplazamiento y reducción producen los siguientes cambios en la configuración:

- Si ACCIÓN$[s_m, a_i] =$ *desplaza* s, entonces la nueva configuración es:

$$(s_0\ X_1\ s_1\ X_2\ s_2 \ldots X_m\ s_m\ a_i\ s\ ,\ a_{i+1}\ \ldots\ a_n\ \$)$$

- Si ACCIÓN$[s_m, a_i] =$ *reduce* $A \to \beta$, la longitud $|\beta|$ es r y la entrada de la tabla IR-A$[s_{m-r}, A] = s$, entonces la nueva configuración es:

$$(s_0\ X_1\ s_1\ X_2\ s_2 \ldots X_{m-r}\ s_{m-r}\ A\ s\ ,\ a_i\ a_{i+1}\ \ldots\ a_n\ \$)$$

Con las definiciones anteriores, se puede plantear el siguiente algoritmo de análisis sintáctico LR.

Algoritmo 3.4. Análisis sintáctico LR.
Entrada: una cadena de entrada w y una tabla de análisis sintáctico LR con las funciones ACCIÓN e IR-A para la gramática G.
Salida: Si $w \in L(G)$, un análisis sintáctico ascendente de w; de lo contrario, se indica *error*.

---

1. Inicialización:
   1.1. La pila contiene s_0.
   1.2. La entrada contiene $w\$$.
   1.3. El apuntador de la entrada ae señala al primer símbolo de $w\$$.
2. REPETIR
   2.1. Sea s el estado en la cima de la pila y a el símbolo apuntado por ae;
   2.2. Si ACCIÓN$[s, a] ==$ *desplazar* s'
      2.2.1. Apilar a y después apilar s';
      2.2.2. Avanzar ae al siguiente símbolo de entrada;
   2.3. en otro caso, si ACCIÓN$[s, a] ==$ *reducir* $A \to \beta$
      2.3.1. Desapilar $2 \times |\beta|$ entradas (símbolos y estados);
      2.3.2. Sea s' el estado que queda en la cima de la pila;
      2.3.3. Apilar A y después apilar el valor de IR-A$[s', A]$;
      2.3.4. Emitir la producción $A \to \beta$;
   2.4. en otro caso, si ACCIÓN$[s, a] ==$ *aceptar*
      2.4.1. Finalizar el análisis;
   2.5. en otro caso
      2.5.1. Informar del error y tratar de continuar el análisis;

Ejemplo 3.8. Antes de estudiar los métodos de construcción de la tabla de análisis, veamos un ejemplo de aplicación del algoritmo anterior. Supongamos que tenemos la conocida gramática:

$$
\begin{array}{ll}
(1) & E \to E + T \\
(2) & E \to T \\
(3) & T \to T * F \\
(4) & T \to F \\
(5) & F \to (E) \\
(6) & F \to id
\end{array}
$$

Cada regla de producción aparece numerada, y nos referiremos a ella mediante el índice correspondiente. Por otra parte, emplearemos los siguientes códigos para representar a las cuatro posibles acciones del autómata:

- di significa desplazar y apilar el estado i,
- rj significa reducir empleando la regla de producción número j,
- acc significa aceptar,
- una casilla en blanco significa error.

En este ejemplo usaremos la siguiente tabla de análisis:

Estado	Acción						Ir-a		
	id	$+$	$*$	$($	$)$	\$	E	T	F
0	d5			d4			1	2	3
1		d6				acc			
2		r2	d7		r2	r2			
3		r4	r4		r4	r4			
4	d5			d4			8	2	3
5		r6	r6		r6	r6			
6	d5			d4				9	3
7	d5			d4					10
8		d6			d11				
9		r1	d7		r1	r1			
10		r3	r3		r3	r3			
11		r5	r5		r5	r5			

La evolución del algoritmo para el análisis de $w = id * id + id$ se muestra en la siguiente tabla:

Pila	Entrada	Acción
0	$id * id + id \$$	d5
$0\ id\ 5$	$* id + id \$$	r6 $F \to id$
$0\ F\ 3$	$* id + id \$$	r4 $T \to F$
$0\ T\ 2$	$* id + id \$$	d7
$0\ T\ 2 * 7$	$id + id \$$	d5
$0\ T\ 2 * 7\ id\ 5$	$+ id \$$	r6 $F \to id$
$0\ T\ 2 * 7\ F\ 10$	$+ id \$$	r3 $T \to T * F$
$0\ T\ 2$	$+ id \$$	r2 $E \to T$
$0\ E\ 1$	$+ id \$$	d6
$0\ E\ 1 + 6$	$id \$$	d5
$0\ E\ 1 + 6\ id\ 5$	\$	r6 $F \to id$
$0\ E\ 1 + 6\ F\ 3$	\$	r4 $T \to F$
$0\ E\ 1 + 6\ T\ 9$	\$	r1 $E \to E + T$
$0\ E\ 1$	\$	acc

Inicialmente, la pila contiene únicamente el estado 0, mientras que la entrada está formada por la cadena $w\$$, siendo id el símbolo actual de la entrada. En esta configuración, podemos comprobar que Acción$[0, id]$ contiene d5,

de modo que para formar la siguiente configuración (segunda línea) se apila el símbolo id y el estado 5, avanzando el puntero de la entrada; por ello, el siguiente terminal a procesar es $*$.

Observamos ahora que ACCIÓN$[5, *]$ contiene r6, de modo que debemos reducir con la regla $F \rightarrow id$ (regla número 6). El pivote es id (lado derecho de la regla), y por tanto sólo sacamos un estado y un símbolo de la pila (secuencia id 5). El estado que queda en la cima es el 0, y ahora apilamos el símbolo F, puesto que es el no terminal del lado izquierdo de la regla de producción que hemos usado para reducir. Esto deja la pila temporalmente con la secuencia 0 F. Para completar la acción debemos apilar el estado indicado por IR-A$[0, F]$, que es 3. Los restantes movimientos se determinan de modo similar.

3.5.4. Tabla de análisis SLR

Comenzamos el estudio de la construcción de tablas de análisis con el método SLR. La construcción de una tabla SLR es relativamente sencilla en comparación con las tablas generadas por los métodos LR-canónico y $LALR$, que se estudian en las secciones siguientes. Sin embargo, el conjunto de gramáticas para las cuales se puede construir este tipo de tablas es el más pequeño de los tres métodos.

Antes de iniciar la descripción del método SLR, introducimos algunas definiciones de utilidad.

Definición 3.19. *Sea $\alpha\beta w$ una forma sentencial derecha (véase la definición 3.7) de una gramática G, siendo β su pivote (véase la definición 3.9), es decir:*

$$S \Rightarrow_{md}^{*} \alpha A w \Rightarrow_{md} \alpha\beta w$$

Se dice que la cadena γ es un prefijo viable *de G si γ es un prefijo de $\alpha\beta$, es decir, si γ es un prefijo de una forma sentencial derecha que no sobrepasa el límite derecho del pivote de dicha forma sentencial.*

Ejemplo 3.9. La derivación más a la derecha de la cadena $id * id + id$ calculada por el algoritmo LR en el ejemplo anterior (3.8) es la siguiente:

$$E \Rightarrow E + T \Rightarrow E + F \Rightarrow E + id \Rightarrow T + id \Rightarrow T * F + id \Rightarrow T * id + id \Rightarrow F * id + id \Rightarrow id * id + id$$

Seleccionando la forma sentencial $T * F + id$, cuyo pivote es $T * F$, tenemos tres prefijos viables: T, $T*$ y $T * F$. Es interesante observar que esta secuencia de prefijos representa el reconocimiento progresivo del lado derecho de la regla de producción $T \rightarrow T * F$.

También es interesante observar que un prefijo viable puede serlo en relación a múltiples formas sentenciales derechas. Por ejemplo, el prefijo T también lo es de las formas sentenciales $T + id$ y $T * id + id$.

Definición 3.20. *Dada una gramática libre de contexto G, llamamos* ítem $LR(0)$ *o simplemente ítem de esta gramática a cualquier producción de la gramática con un punto en su lado derecho.*

Ejemplo 3.10. Dada la gramática del ejemplo 3.8, la producción $E \rightarrow E + T$ permite construir cuatro ítems:

$$E \rightarrow \cdot E + T$$
$$E \rightarrow E \cdot + T$$
$$E \rightarrow E + \cdot T$$
$$E \rightarrow E + T \cdot$$

Si tenemos una regla del tipo $A \rightarrow \lambda$, sólo es posible construir un ítem de la forma $A \rightarrow \cdot$.

Los ítems son la herramienta básica para la construcción del autómata de pila. El punto de un ítem nos permite representar la porción de una regla que ha sido reconocida en la entrada. Por ejemplo, el ítem

$E \to E \cdot +T$ indica que acabamos de reconocer una cadena derivable de E y ahora esperamos encontrar en la entrada una cadena derivable de $+T$. Como el símbolo a la derecha de un punto es el terminal $+$, el ítem está simbolizando la acción de desplazamiento al encontrar este terminal.

El ítem $E \to E + \cdot T$ representa la situación a la que se llega partiendo del ítem anterior al encontrar el símbolo $+$. Ahora el símbolo a la derecha del punto es el no terminal T. Por tanto, el ítem simboliza una situación en la que se espera encontrar una cadena derivable de T en la entrada. Pero precisamente esta circunstancia viene representada por los ítems $T \to \cdot T * F$ y $T \to \cdot F$. Por tanto, un ítem con no terminal a la derecha del punto está vinculado con ítems en los que dicho no terminal aparece en el lado izquierdo, y el punto al comienzo del lado derecho.

Cuando el punto está en el extremo derecho, como en el ítem $E \to E + T\cdot$, indicamos la situación en la que se ha reconocido una cadena completa derivable del lado derecho de una producción. Por tanto, este ítem actúa como un indicador de que podemos aplicar una reducción empleando como pivote el lado derecho de esta regla de producción.

Todos los ítems $LR(0)$ posibles de una gramática se pueden agrupar en conjuntos, denominados colección $LR(0)$. Los ítems incluidos en un mismo conjunto deben representar una situación coherente del autómata, es decir, uno de sus estados. Los ítems de un estado permiten deducir las acciones aplicables en dicho estado.

Para construir la colección $LR(0)$ es necesario definir una extensión de la gramática G y dos operaciones denominadas CLAUSURA y GOTO.

Definición 3.21. *Dada una gramática G libre de contexto con símbolo inicial S, su* gramática aumentada *G' se construye añadiendo el nuevo símbolo terminal inicial S' y la regla de producción $S' \to S$.*

El propósito de esta nueva regla es indicar al analizador cuándo debe finalizar el análisis e indicar la aceptación de la entrada. Esto sucederá al reducir esta regla de producción.

Si I es un conjunto de ítems de una gramática G, entonces la CLAUSURA(I) es el conjunto de ítems construido con el siguiente algoritmo.

Algoritmo 3.5. Cálculo de CLAUSURA(I).
Entrada: una gramatica libre de contexto G y un conjunto de ítems LR(0) I de dicha gramática.
Salida: conjunto de ítems CLAUSURA(I).

---

1. $J = I$;
2. REPETIR
   Por cada elemento $A \to \alpha \cdot B\beta$ en J
   Por cada producción $B \to \gamma$ de G
   Si $B \to \cdot\gamma$ no está en J, añadir $B \to \cdot\gamma$ a J;
   HASTA que no se puedan añadir más elementos a J;
3. DEVOLVER J;

Ejemplo 3.11. Dada la gramática del ejemplo 3.8, si $I = \{E \to \cdot E + T\}$ entonces la CLAUSURA(I) es:

$$\begin{aligned}\text{CLAUSURA}(I) = \{&E \to \cdot E + T\\ &E \to \cdot T\\ &T \to \cdot T * F\\ &T \to \cdot F\\ &F \to \cdot (E)\\ &F \to \cdot id\ \}\end{aligned}$$

Dado un conjunto de ítems I, y un símbolo de la gramática no extendida $X \in \{V_N \cup V_T\}$, se define GOTO(I, X) = CLAUSURA(I_X) siendo $I_X = \{A \to \alpha X \cdot \beta \mid A \to \alpha \cdot X\beta \in I\}$. Es decir, en primer lugar se desplaza el punto a la derecha de X en todos aquellos ítems de I que tengan el punto a su izquierda, y posteriormente se aplica la operación de CLAUSURA. Obsérvese que GOTO(I, X) puede ser el conjunto vacío en el caso de que el conjunto de ítems I no tenga ninguno con X a la derecha del punto.

Ejemplo 3.12. Dada la gramática del ejemplo 3.8, supongamos que $I = \{E \to T\cdot\ ,\ T \to T \cdot *F\}$, entonces el conjunto GOTO$(I, *)$ es:

$$\begin{aligned}\text{GOTO}(I, *) = \{&T \to T * \cdot F\\ &F \to \cdot (E)\\ &F \to \cdot id\ \}\end{aligned}$$

El primer ítem se obtiene desplazando el punto a la derecha de $*$, mientras que los dos últimos se obtienen por aplicación de la operación CLAUSURA.

Una vez definidas las dos operaciones CLAUSURA y GOTO, es posible construir la colección $LR(0)$ completa. Para ello se emplea la gramática extendida G', aplicando el siguiente algoritmo.

Algoritmo 3.6. Cálculo de la colección $LR(0)$.
Entrada: una gramatica libre de contexto extendida G'.
Salida: colección $LR(0)$.

---

1. $C =$ CLAUSURA$(\{S' \to \cdot S\})$;
2. REPETIR

   Por cada conjunto de elementos I en C

   Por cada símbolo gramatical X

   Si GOTO(I, X) no está vacío y no está en C , añadir GOTO(I, X) a C;

   HASTA que no se puedan añadir más conjuntos de elementos a C;
3. DEVOLVER C;

Ejemplo 3.13. Aplicamos el algoritmo 3.6 a la gramática extendida:

$$
\begin{aligned}
&E' \to E\\
&E \to E + T \mid T\\
&T \to T * F \mid F\\
&F \to (E) \mid id
\end{aligned}
$$

$$
\begin{aligned}
I_0 = \textsc{Clausura}(\{E' \to \cdot E\}) = \{&E' \to \cdot E\\
&E \to \cdot E + T\\
&E \to \cdot T\\
&T \to \cdot T * F\\
&T \to \cdot F\\
&F \to \cdot (E)\\
&F \to \cdot id\ \}
\end{aligned}
$$

$$
\begin{aligned}
\textsc{Goto}(I_0, E) = I_1 = \{\ &E' \to E\cdot\\
&E \to E \cdot + T\ \}
\end{aligned}
$$

$$
\begin{aligned}
\textsc{Goto}(I_0, T) = I_2 = \{\ &E \to T\cdot\\
&T \to T \cdot * F\ \}
\end{aligned}
$$

$$
\textsc{Goto}(I_0, F) = I_3 = \{\ T \to F\cdot\ \}
$$

$$
\begin{aligned}
\textsc{Goto}(I_0, () = I_4 = \{\ &F \to (\cdot E)\\
&E \to \cdot E + T\\
&E \to \cdot T\\
&T \to \cdot T * F\\
&T \to \cdot F\\
&F \to \cdot (E)\\
&F \to \cdot id\ \}
\end{aligned}
$$

$$
\textsc{Goto}(I_0, id) = I_5 = \{\ F \to id\cdot\ \}
$$

$$
\begin{aligned}
\textsc{Goto}(I_1, +) = I_6 = \{\ &E \to E + \cdot T\\
&T \to \cdot T * F\\
&T \to \cdot F\\
&F \to \cdot (E)\\
&F \to \cdot id\ \}
\end{aligned}
$$

$$
\begin{aligned}
\textsc{Goto}(I_2, *) = I_7 = \{\ &T \to T * \cdot F\\
&F \to \cdot (E)\\
&F \to \cdot id\ \}
\end{aligned}
$$

$$
\begin{aligned}
\textsc{Goto}(I_4, E) = I_8 = \{\ &F \to (E\cdot)\\
&E \to E \cdot + T\ \}
\end{aligned}
$$

$$
\textsc{Goto}(I_4, T) = I_2
$$

$$
\textsc{Goto}(I_4, F) = I_3
$$

$$
\textsc{Goto}(I_4, () = I_4
$$

$$
\textsc{Goto}(I_4, id) = I_5
$$

$$
\begin{aligned}
\textsc{Goto}(I_6, T) = I_9 = \{\ &E \to E + T\cdot\\
&T \to T \cdot * F\ \}
\end{aligned}
$$

$$
\textsc{Goto}(I_6, F) = I_3
$$

$$
\textsc{Goto}(I_6, () = I_4
$$

$$
\textsc{Goto}(I_6, id) = I_5
$$

$$
\textsc{Goto}(I_7, F) = I_{10} = \{\ T \to T * F\cdot\ \}
$$

$$
\textsc{Goto}(I_7, () = I_4
$$

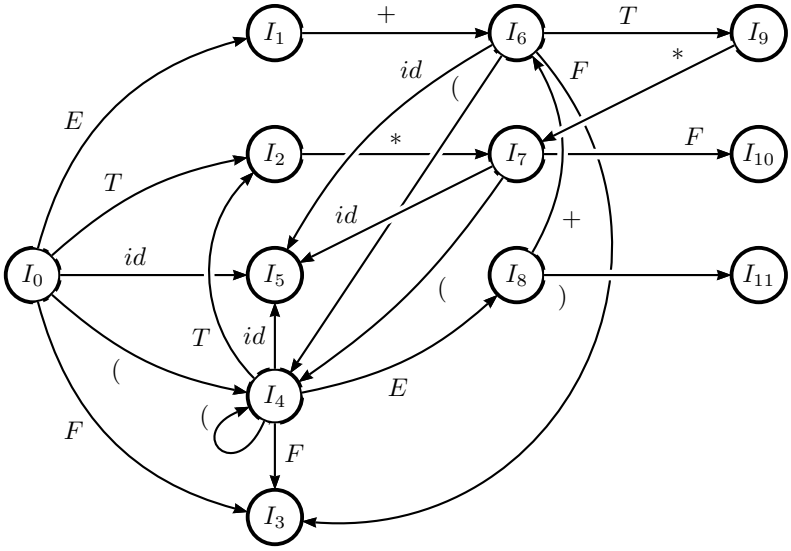
$$
\textsc{Goto}(I_7, id) = I_5
$$

$$
\textsc{Goto}(I_8,)) = I_{11} = \{\ F \to (E)\cdot\ \}
$$

$$
\textsc{Goto}(I_8, +) = I_6
$$

$$
\textsc{Goto}(I_9, *) = I_7
$$

Obsérvese que algunas de las aplicaciones de la operación GOTO no conducen a la construcción de conjuntos de ítems nuevos, sino que vuelven a producir algunos de los ya existentes. Si tratamos a estos conjuntos de ítems como estados, y a la operación GOTO como la función de transición, obtenemos el siguiente autómata:



Este autómata reconoce los prefijos viables de la gramática. Como vimos en el ejemplo 3.9, la forma sentencial derecha $T*F+id$ tiene los prefijos viables T, $T*$ y $T*F$. Si, partiendo del estado inicial I_0, seguimos las transiciones etiquetadas con los símbolos T, $*$ y F, encontraremos estados en los que hay ítems que indican el reconocimiento de cada uno de estos prefijos. I_2, al que se llega desde I_0 con la transición T, contiene el ítem $T \to T \cdot *F$; I_7, al que se llega con las transiciones T y $*$, contiene el ítem $T \to T * \cdot F$; I_{10}, al que se llega con las transiciones T, $*$ y F, contiene el ítem $T \to T * F\cdot$.

Definición 3.22. *Decimos que un ítem* $LR(0)$ $A \to \beta_1 \cdot \beta_2$ *es* válido *para un prefijo viable* $\alpha\beta_1$*, si existe una derivación* $S' \Rightarrow^*_{md} \alpha A w \Rightarrow_{md} \alpha\beta_1\beta_2 w$ *donde* $w \in V_T^*$ *y* $\alpha, \beta_1, \beta_2 \in (V_T \cup V_N)^*$.

Se puede comprobar que el conjunto de ítems válidos para un prefijo viable γ es el conjunto I_i de la colección que corresponde al estado i-ésimo del autómata. Este estado se alcanza partiendo desde el estado inicial y recorriendo el camino dado por los símbolos de γ.

Ejemplo 3.14. Tomando el autómata del ejemplo anterior, podemos observar que $E + T*$ es un prefijo viable, ya que todas las transiciones desde I_0 están definidas. Partiendo de I_0, y siguiendo los símbolos del prefijo, el autómata entra en el estado I_7, que contiene:

$$T \to T * \cdot F$$
$$F \to \cdot(E)$$
$$F \to \cdot id$$

Estos son ítems válidos para el prefijo $\alpha\beta_1 = E + T*$, ya que:

$$E' \Rightarrow^*_{md} \underbrace{E+}_{\alpha} \underbrace{T}_{w} \Rightarrow_{md} E + \underbrace{T*}_{\beta_1}\underbrace{F}_{\beta_2} \quad \text{para el ítem } T \to T * \cdot F$$

$$E' \Rightarrow^*_{md} \underbrace{E + T*}_{\alpha} \underbrace{F}_{w} \Rightarrow_{md} E + T * \underbrace{}_{\beta_1}\underbrace{(E)}_{\beta_2} \quad \text{para el ítem } F \to \cdot(E)$$

$$E' \Rightarrow^*_{md} \underbrace{E + T*}_{\alpha} \underbrace{F}_{w} \Rightarrow_{md} E + T * \underbrace{}_{\beta_1}\underbrace{id}_{\beta_2} \quad \text{para el ítem } F \to \cdot id$$

El hecho de que el ítem $A \to \beta_1 \cdot \beta_2$ sea válido para el prefijo $\alpha\beta_1$ contribuye a decidir si se debe desplazar o reducir cuando se encuentra el prefijo $\alpha\beta_1$ en la pila, según sea β_2. Si $\beta_2 \neq \lambda$, todavía no se

ha introducido completamente el pivote en la pila, de modo que puede interesar desplazar símbolos de la entrada hasta alcanzar el extremo derecho de β_2. Si por el contrario $\beta_2 = \lambda$, entonces la pila tiene en su cima el pivote completo, y puede haber llegado el momento de reducir.

No obstante, la decisión entre desplazar o reducir no se adopta únicamente por la forma de los ítems válidos del estado al que conduce el prefijo viable. En el autómata del ejemplo 3.13 encontramos el estado I_9 con los ítems $E \to E + T\cdot$ y $T \to T \cdot *F$. El primero parece indicar que hay que reducir, mientras que el segundo apunta a que hay que desplazar.

La decisión de la acción a realizar se toma con la ayuda del carácter de anticipación. Si el siguiente token es el símbolo $*$, se debe realizar la acción de desplazar, que simboliza el segundo ítem. La acción de reducir se puede aplicar al encontrar en la entrada un token perteneciente al conjunto SIGUIENTE(E), ya que E está en el lado izquierdo del ítem $E \to E + T\cdot$ que se usa para reducir.

Con todo lo anterior se puede plantear el siguiente algoritmo para construir la tabla de análisis SLR.

Algoritmo 3.7. Construcción de una tabla de análisis sintáctico SLR.
Entrada: una gramática libre de contexto aumentada G'.
Salida: las tablas ACCIÓN e IR-A para el análisis SLR de G'.

---

1. Constrúyase la colección $LR(0)$ $C = \{I_0, I_1, \ldots, I_n\}$ para G', usando el algoritmo 3.6;
2. El estado i se construye a partir de I_i. Las acciones de análisis sintáctico para el estado se determinan como sigue:
   a) Si $A \to \alpha \cdot a\beta \in I_i$, siendo $a \in V_T$ y GOTO$(I_i, a) = I_j$ entonces
   asígnese **desplazar** j a ACCIÓN$[i, a]$.
   b) Si $A \to \alpha\cdot \in I_i$, y $A \neq S'$, entonces
   asígnese **reducir** $A \to \alpha$ a ACCIÓN$[i, a]$ para todo $a \in$ SIGUIENTE(A).
   c) Si $S' \to S\cdot \in I_i$, entonces
   asígnese **aceptar** a ACCIÓN$[i, \$]$.

   Si las reglas anteriores no generan acciones contradictorias, se dice que la gramática es $SLR(1)$. En caso contrario, el algoritmo no consigue producir un analizador sintáctico.
3. Las transiciones IR-A para el estado i se construyen para todos los no terminales A utilizando la regla: Si GOTO$(I_i, A) = I_j$ entonces IR-A$[i, A] = j$.
4. Todas las entradas no definidas por las reglas 2 y 3 son consideradas **error**.
5. El estado inicial del analizador es el construido a partir del ítem $S' \to \cdot S$.

Ejemplo 3.15. Aplicamos el algoritmo anterior para la construcción de la tabla de análisis SLR de la gramática:

$$E \to E + T \mid T$$
$$T \to T * F \mid F$$
$$F \to (E) \mid id$$

Partiendo de la colección de ítems obtenida en el ejercicio 3.13, calculamos la tabla de análisis de la gramática

para las expresiones aritméticas. Los conjuntos PRIMERO y SIGUIENTE de la gramática son los siguientes:

$$\begin{aligned}\text{PRIMERO}(E) &= \{(, id\}\\ \text{PRIMERO}(T) &= \{(, id\}\\ \text{PRIMERO}(F) &= \{(, id\}\\ \text{SIGUIENTE}(E) &= \{\$, +,)\}\\ \text{SIGUIENTE}(T) &= \{\$, +,), *\}\\ \text{SIGUIENTE}(F) &= \{\$, +,), *\}\end{aligned}$$

La tabla de análisis obtenida es la que se empleó en el ejercicio 3.8:

ESTADO	ACCIÓN						IR-A		
	id	$+$	$*$	$($	$)$	\$	E	T	F
0	d5			d4			1	2	3
1		d6				acc			
2		r2	d7		r2	r2			
3		r4	r4		r4	r4			
4	d5			d4			8	2	3
5		r6	r6		r6	r6			
6	d5			d4				9	3
7	d5			d4					10
8		d6			d11				
9		r1	d7		r1	r1			
10		r3	r3		r3	r3			
11		r5	r5		r5	r5			

La tabla no contiene conflictos, por lo que se deduce que la gramática es de tipo SLR. Veamos cómo se generan las entradas del estado 0 a partir del conjunto de ítems I_0:

$$\begin{aligned}I_0 = \{ &[\, E' \to \cdot E\,],\ [\, E \to \cdot E + T\,],\ [\, E \to \cdot T\,],\ [\, T \to \cdot T * F\,],\\ &[\, T \to \cdot F\,],\ [\, F \to \cdot(E)\,],\ [\, F \to \cdot id\,]\ \}\end{aligned}$$

Los ítems que tienen el punto a la izquierda de un no terminal producen entradas en la sección IR-A. Por ejemplo, al ser GOTO$(I_0, E) = I_1$, la entrada IR-A$[0, E]$ toma el valor 1, indicando que el reconocimiento del no terminal E en el estado 0 conduce al estado 1. De forma similar se obtienen las entradas IR-A$[0, T]$ e IR-A$[0, F]$. Por otro lado, los ítems que tienen el punto a la izquierda de un terminal producen entradas en la sección ACCIÓN. En concreto, generan acciones de desplazamiento. Por ejemplo, al ser GOTO$(I_0, id) = I_5$, la entrada ACCIÓN$[0, id]$ toma el valor d5, indicando que, al detectar el terminal id en la entrada, hallándose el autómata en el estado 0, se debe desplazar dicho terminal a la pila y pasar al estado 5. La deducción para el terminal (es similar.

Observemos ahora los ítems del conjunto I_2, que determinan las entradas del estado 2:

$$I_2 = \{\ [\, E \to T\cdot\,],\ [\, T \to T \cdot *F\,]\ \}$$

Los ítems que tienen el punto al final del lado derecho conducen a acciones de reducción. En concreto, el ítem $[\, E \to T\cdot\,]$, construido a partir de la regla de producción número 2, indica que, hallándose el autómata en el estado 2, si la entrada es cualquiera de los tokens pertenecientes al conjunto SIGUIENTE(E), la acción debe ser r2, es decir, una reducción empleando la regla número 2. Como el conjunto SIGUIENTE(E) está formado por los terminales $\{\$, +,)\}$, la acción r2 debe asignarse a las entradas ACCIÓN$[2, \$]$, ACCIÓN$[2, +]$ y ACCIÓN$[2,)]$. Por otro lado, el ítem $[\, T \to T \cdot *F\,]$ genera una acción de desplazamiento al encontrar el token $*$ en la entrada, pasando al estado 7 ya que GOTO$[2, *] = I_7$.

Finalmente, el conjunto I_1 es:

$$I_1 = \{\ [\, E' \to E\cdot\,],\ [\, E \to E \cdot +T\,]\ \}$$

El ítem $[\, E' \to E\cdot\,]$ es especial, ya que supone la reducción de la regla $E' \to E$ con la que se ha extendido la gramática original. Este ítem genera la acción de aceptación al encontrar el token \$ en la entrada, es decir, ACCIÓN$[1, \$]$ = acc. El ítem $[\, E \to E \cdot +T\,]$ produce una acción d6, debido a que GOTO$[1, +] = I_6$.

Los restantes conjuntos de la colección $LR(0)$ producen, de forma similar, las demás acciones reflejadas en la tabla.

La condición de partida para plantear la construcción de la tabla SLR es que la gramática no sea ambigua. Toda gramática ambigua generará conflictos en la tabla de análisis. Sin embargo, podemos encontrar gramáticas que no son ambiguas pero que producen conflictos en la tabla de análisis SLR. El siguiente ejemplo muestra un caso.

Ejemplo 3.16. Considérese la gramática siguiente, que representa asignaciones con punteros:

$$\begin{array}{ll} (1) & S \to L = R \\ (2) & S \to R \\ (3) & L \to *R \\ (4) & L \to id \\ (5) & R \to L \end{array}$$

La aplicación del algoritmo 3.6 genera la siguiente colección de ítems $LR(0)$:

$$\begin{aligned} I_0 = \{\ & S' \to \cdot S \\ & S \to \cdot L = R \\ & S \to \cdot R \\ & L \to \cdot * R \\ & L \to \cdot id \\ & R \to \cdot L\ \} \end{aligned}$$

$$\text{GOTO}(I_0, S) = I_1 = \{\ S' \to S\cdot\ \}$$

$$\begin{aligned} \text{GOTO}(I_0, L) = I_2 = \{\ & S \to L\cdot = R \\ & R \to L\cdot\ \} \end{aligned}$$

$$\text{GOTO}(I_0, R) = I_3 = \{\ S \to R\cdot\ \}$$

$$\begin{aligned} \text{GOTO}(I_0, *) = I_4 = \{\ & L \to * \cdot R \\ & R \to \cdot L \\ & L \to \cdot * R \\ & L \to \cdot id\ \} \end{aligned}$$

$$\text{GOTO}(I_0, id) = I_5 = \{\ L \to id\cdot\ \}$$

$$\begin{aligned} \text{GOTO}(I_2, =) = I_6 = \{\ & S \to L = \cdot R \\ & R \to \cdot L \\ & L \to \cdot * R \\ & L \to \cdot id\ \} \end{aligned}$$

$$\text{GOTO}(I_4, R) = I_7 = \{\ L \to *R\cdot\ \}$$

$$\text{GOTO}(I_4, L) = I_8 = \{\ R \to L\cdot\ \}$$

$$\text{GOTO}(I_4, *) = I_4$$

$$\text{GOTO}(I_4, id) = I_5$$

$$\text{GOTO}(I_6, R) = I_9 = \{S \to L = R\cdot\ \}$$

$$\text{GOTO}(I_6, L) = I_8$$

$$\text{GOTO}(I_6, *) = I_4$$

$$\text{GOTO}(I_6, id) = I_5$$

Observemos el conjunto de ítems I_2. Teniendo en cuenta que el conjunto SIGUIENTE(R) es $\{ =, \$ \}$, I_2 implica dos acciones que provocan un conflicto:

- El ítem $[\ S \to L\cdot = R\ ]$ genera una acción de *desplazamiento* al encontrar el símbolo $=$.
- El ítem $[\ R \to L\cdot\ ]$ genera una acción de *reducción* al encontrar los símbolos $=$ y $\$$.

Por tanto, hay un conflicto desplaza/reduce cuando se encuentra el signo $=$ en el estado 2. La gramática propuesta no es ambigua, de modo que el conflicto surge por las propias limitaciones del método SLR para construir la tabla de análisis. Obsérvese que no existe en el lenguaje generado por la gramática una secuencia que comience con $R = \ldots$, por lo que la acción de reducción no debería considerarse en este estado.

El problema del método SLR mostrado en el ejemplo anterior radica en que puede haber ciertos prefijos viables para los cuales no sea correcta una reducción. El método SLR carece de la potencia necesaria para recordar el contexto a la izquierda del punto. Para superar esta limitación, se debe dotar a los estados de información adicional que permita detectar y prohibir este tipo de reducciones no válidas.

3.5.5. Tabla de análisis LR-canónica

Para añadir más información al análisis, se debe redefinir el concepto de ítem, añadiendo un segundo componente al mismo que aporta la información necesaria para evitar las reducciones no válidas.

Definición 3.23. *Llamamos ítem $LR(1)$ de una gramática aumentada G' a un elemento de la forma:*

$$[A \to \alpha \cdot \beta, a]$$

donde $A \to \alpha\beta \in P$ y $a \in (V_T \cup \{\$\})$.

El número 1 denota la longitud del segundo componente del ítem, un símbolo de anticipación. Este símbolo no va a tener efecto en aquellos ítems en los cuales $\beta \neq \lambda$. Sin embargo, un ítem $[A \to \alpha\cdot, a]$ indicará que se aplique una reducción sólo en el caso de que el siguiente símbolo de la entrada sea a. Lógicamente, los símbolos de anticipación a formarán un subconjunto de SIGUIENTE(A). Esta restricción de la reducción a un subconjunto de SIGUIENTE(A) permitirá evitar reducciones no válidas.

La nueva definición de ítem $LR(1)$ implica también una revisión del concepto de ítem válido.

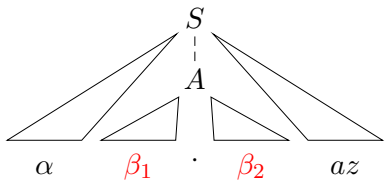
Definición 3.24. *Decimos que un ítem $LR(1)$ $[A \to \beta_1 \cdot \beta_2, a]$ es válido para el prefijo viable $\alpha\beta_1$ si:*

$$S' \Rightarrow^*_{md} \alpha A w \Rightarrow_{md} \alpha\beta_1\beta_2 w$$

siendo $\alpha, \beta_1, \beta_2 \in (V_T \cup V_N)^*$ y $w \in V_T^*$, cumpliéndose además una de las dos condiciones siguientes:

- *Si $w \neq \lambda$, a es el primer símbolo de w.*
- *Si $w = \lambda$, a es el marcador final \$.*

La situación descrita por la definición anterior se puede reflejar en el siguiente árbol de análisis ($w = az$):



El procedimiento de construcción de la colección de conjuntos de elementos $LR(1)$ válidos es muy similar al que vimos en la sección anterior para la colección $LR(0)$. Los cambios con respecto a aquél se localizan en las definiciones de las operaciones CLAUSURA y GOTO, que se indican a continuación.

Algoritmo 3.8. Cálculo de CLAUSURA(I).
Entrada: una gramatica libre de contexto G y un conjunto de ítems $LR(1)$ I de dicha gramática.
Salida: conjunto de ítems CLAUSURA(I).

---

1. $J = I$;
2. REPETIR

   Por cada elemento $[A \to \alpha \cdot B\beta, a]$ en J

   Por cada producción $B \to \gamma$ de G

   Por cada terminal $b \in$ PRIMERO(βa)

   Si $[B \to \cdot\gamma, b]$ no está en J, añadirlo a J;

   HASTA que no se puedan añadir más ítems a J;
3. DEVOLVER J;

Ejemplo 3.17. Considérese la gramática usada en el ejemplo 3.16. Supongamos que disponemos del conjunto de ítems $I = \{\ [S \to \cdot L = R, \$]\ \}$. Según la definición anterior, CLAUSURA(I) es:

$$\begin{aligned}\text{CLAUSURA}(I) = \{&[S \to \cdot L = R, \$]\\ &[L \to \cdot * R, =]\\ &[L \to \cdot id, =]\ \}\end{aligned}$$

El signo de anticipación $=$ del segundo y tercer ítem se obtienen del cálculo de PRIMERO$(= R\$)$.

La modificación en la definición de ítem nos obliga a adaptar también la definición de la función GOTO. Dado un conjunto de ítems $LR(1)$ I, y un símbolo de la gramática no extendida $X \in \{V_N \cup V_T\}$, se define GOTO(I, X) = CLAUSURA(I_X) siendo $I_X = \{[A \to \alpha X \cdot \beta, a] \mid [A \to \alpha \cdot X\beta, a] \in I\}$. Es decir, la operación no cambia en esencia con respecto a los ítems $LR(0)$.

Una vez actualizadas las dos operaciones CLAUSURA y GOTO, es posible construir la colección $LR(1)$ completa. Para ello se emplea la gramática extendida G', aplicando el siguiente algoritmo.

Algoritmo 3.9. Cálculo de la colección $LR(1)$.
Entrada: una gramatica libre de contexto extendida G'.
Salida: colección $LR(1)$.

---

1. $C =$ CLAUSURA$(\{[S' \to \cdot S, \$]\})$;
2. REPETIR

   Por cada conjunto de elementos I en C

   Por cada símbolo gramatical X

   Si GOTO(I, X) no está vacío y no está en C, añadir GOTO(I, X) a C;

   HASTA que no se puedan añadir más conjuntos de elementos a C;
3. DEVOLVER C;

Ejemplo 3.18. Aplicamos el algoritmo de construcción de la colección $LR(1)$ a la siguiente gramática aumentada:

$$\begin{aligned} S' &\to S \\ S &\to CC \\ C &\to cC \\ C &\to d \end{aligned}$$

En primer lugar se debe calcular $I_0 = \text{CLAUSURA}(\{[S' \to \cdot S, \$]\})$:

- El ítem inicial $[S' \to \cdot S, \$]$ se añade a I_0.
- Teniendo en cuenta que PRIMERO$(\$) = \{\$\}$, debemos introducir en I_0 el nuevo ítem $[S \to \cdot CC, \$]$.
- A partir de este último, considerando que PRIMERO$(C\$) =$ PRIMERO$(C) = \{c, d\}$, podemos añadir cuatro items nuevos: $[C \to \cdot cC, c]$, $[C \to \cdot cC, d]$, $[C \to \cdot d, c]$, $[C \to \cdot d, d]$

Por tanto, el primero conjunto de ítems completo queda de la siguiente forma:

$$\begin{aligned} I_0 = \{\ & [S' \to \cdot S, \$] \\ & [S \to \cdot CC, \$] \\ & [C \to \cdot cC, c/d] \\ & [C \to \cdot d, c/d]\ \} \end{aligned}$$

Obsérvese que se ha simplificado la notación de los últimos cuatro ítems utilizando una notación reducida en la que se representan varios ítems con la misma producción punteada por un único ítem con varios símbolos de anticipación separados por $/$.

Ahora se debe calcular el conjunto $I_1 = \text{GOTO}(I_0, S)$:

$$\text{GOTO}(I_0, S) = I_1 = \{\ [S' \to S\cdot, \$]\ \}$$

El siguiente conjunto I_2 se obtiene con la operación $\text{GOTO}(I_0, C)$. En primer lugar se desplaza el punto a la derecha del símbolo C en los ítems de I_0, llegando al conjunto $\{[S \to C \cdot C, \$]\}$. A continuación se aplica la CLAUSURA sobre este conjunto. Obsérvese que debemos generar nuevos ítems con símbolos de anticipación en el conjunto PRIMERO$(\$) = \{\$\}$:

$$\begin{aligned} \text{GOTO}(I_0, C) = I_2 = \{\ & [S \to C \cdot C, \$] \\ & [C \to \cdot cC, \$] \\ & [C \to \cdot d, \$]\ \} \end{aligned}$$

Seguidamente se obtiene $I_3 = \text{GOTO}(I_0, c)$. El desplazamiento del punto nos lleva al conjunto $\{[C \to c \cdot C, c/d]\}$. Al calcular la CLAUSURA de este conjunto debemos emplear los valores de PRIMERO(c) y PRIMERO(d) como símbolos de anticipación, es decir, los propios símbolos c y d:

$$\begin{aligned} \text{GOTO}(I_0, c) = I_3 = \{\ & [C \to c \cdot C, c/d] \\ & [C \to \cdot cC, c/d] \\ & [C \to \cdot d, c/d]\ \} \end{aligned}$$

Los restantes conjuntos de ítems se obtienen de forma similar:

$$\text{GOTO}(I_0, d) = I_4 = \{\ [C \to d\cdot, c/d]\ \}$$

$$\text{GOTO}(I_2, C) = I_5 = \{\ [S \to CC\cdot, \$]\ \}$$

$$\begin{aligned} \text{GOTO}(I_2, c) = I_6 = \{\ & [C \to c \cdot C, \$] \\ & [C \to \cdot cC, \$] \\ & [C \to \cdot d, \$]\ \} \end{aligned}$$

$$\text{GOTO}(I_2, d) = I_7 = \{\ [C \to d\cdot, \$]\ \}$$

$$\text{GOTO}(I_3, C) = I_8 = \{\ [C \to cC\cdot, c/d]\ \}$$

$$\text{GOTO}(I_3, c) = I_3$$

$$\text{GOTO}(I_3, d) = I_4$$

$$\text{GOTO}(I_6, C) = I_9 = \{\ [C \to cC\cdot, \$]\ \}$$

$$\text{GOTO}(I_6, c) = I_6$$

$$\text{GOTO}(I_6, d) = I_7$$

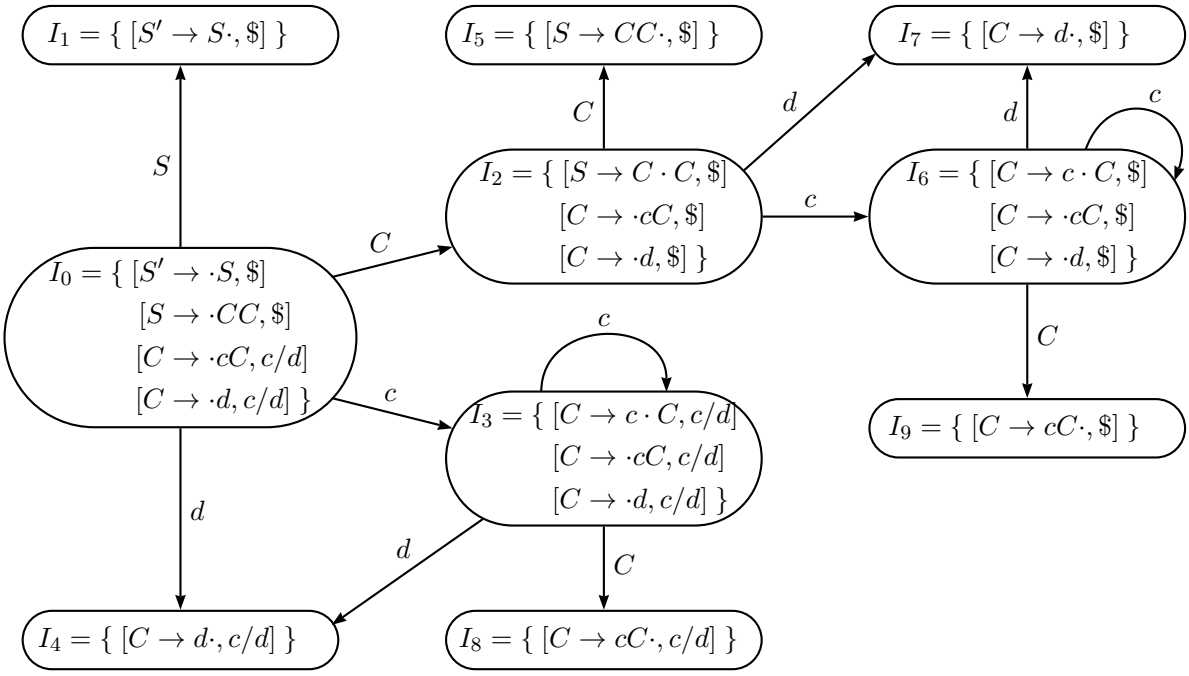


Figura 3.3: Autómata LALR de la gramática del ejemplo 3.18

El autómata generado para el reconocimiento de los prefijos viables es el mostrado en la figura 3.3.

La tabla de análisis LR-canónica tiene una estructura idéntica a la SLR. Lógicamente, dada la nueva definición de ítem $LR(1)$, existen algunas modificaciones en el algoritmo que construye la subtabla ACCIÓN, y más concretamente en la generación de las acciones de reducción.

Algoritmo 3.10. Construcción de una tabla de análisis sintáctico LR-canónica.
Entrada: una gramática libre de contexto aumentada G'.
Salida: las tablas ACCIÓN e IR-A para el análisis LR-canónico de G'.

---

1. Constrúyase la colección $LR(1)$ $C = \{I_0, I_1, \ldots, I_n\}$ para G', usando el algoritmo 3.9;
2. El estado i se construye a partir de I_i. Las acciones de análisis sintáctico para el estado se determinan como sigue:
   a) Si $[A \to \alpha \cdot a\beta, b] \in I_i$, siendo $a \in V_T$ y GOTO$(I_i, a) = I_j$ entonces asígnese **desplazar** j a ACCIÓN$[i, a]$.
   b) Si $[A \to \alpha\cdot, a] \in I_i$, y $A \neq S'$, entonces asígnese **reducir** $A \to \alpha$ a ACCIÓN$[i, a]$.
   c) Si $[S' \to S\cdot, \$] \in I_i$, entonces asígnese **aceptar** a ACCIÓN$[i, \$]$.

   Si las reglas anteriores no generan acciones contradictorias, se dice que la gramática es $LR(1)$. En caso contrario, el algoritmo no consigue producir un analizador sintáctico.
3. Las transiciones IR-A para el estado i se construyen para todos los no terminales A utilizando la regla: Si GOTO$(I_i, A) = I_j$ entonces IR-A$[i, A] = j$.

4. Todas las entradas no definidas por las reglas 2 y 3 son consideradas **error**.
5. El estado inicial del analizador es el construido a partir del ítem $[S' \to \cdot S, \$]$.

Ejemplo 3.19. Si aplicamos el algoritmo anterior a la colección de ítems $LR(1)$ obtenida en el ejercicio 3.18, obtendremos la tabla de análisis LR-canónica que se muestra a continuación:

ESTADO	ACCIÓN			IR-A	
	c	d	\$	S	C
0	d3	d4		1	2
1			acc		
2	d6	d7			5
3	d3	d4			8
4	r3	r3			
5			r1		
6	d6	d7			9
7			r3		
8	r2	r2			
9			r2		

donde las reglas de la gramática han sido numeradas de la siguiente forma:

$$\begin{array}{ll} (1) & S \to CC \\ (2) & C \to cC \\ (3) & C \to d \end{array}$$

Por ejemplo, las acciones de reducción con la regla 3 en el estado 4 se deben al ítem $[C \to d\cdot, c/d]$ que contiene dicho estado. Las acciones aparecen en las columnas c y d, por ser estos los símbolos indicados en el segundo componente del ítem[6].

3.5.6. Tabla de análisis $LALR$

El método $LALR$ para la construcción de tablas de análisis representa un punto intermedio entre las características de los dos métodos vistos hasta ahora. El método SLR consigue una tabla de análisis con una cantidad reducida de estados, pero no es capaz de manejar cierto tipo de gramáticas no ambiguas. El método LR-canónico es más potente, pero en contrapartida suele generar una cantidad mayor de estados, ya que maneja información más detallada en sus ítems.

El método $LALR$ que se estudia en esta sección tiene una potencia intermedia entre el método LR-canónico y el SLR (véase la figura 3.4). No obstante, consigue una tabla de análisis de igual tamaño a la SLR. Esto hace que sea un método muy utilizado en la construcción de compiladores[7].

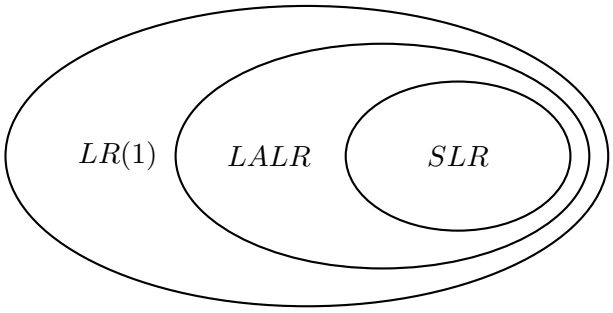


Figura 3.4: Relación entre las gramáticas $LR(1)$, $LALR$ y SLR.

Para construir una tabla de análisis $LALR$, se debe obtener en primer lugar la colección de conjuntos de ítems $LR(1)$, siguiendo el mismo procedimiento que en el caso del método LR-canónico. Sin embargo, en lugar de aplicar a continuación el algoritmo 3.10 para la construcción de la tabla, se realiza un paso previo mediante el cual se reduce la cantidad de estados del autómata que reconoce los prefijos viables.

Si se observa el autómata generado en el ejemplo 3.18 se puede comprobar que existen estados muy parecidos en cuanto a los ítems que contienen. En concreto, hay algunos estados que son idénticos en los

[6] Recuérdese que, realmente, este ítem está representando de forma simplificada a dos ítems: $[C \to d\cdot, c]$ y $[C \to d\cdot, d]$.
[7] Por defecto, la herramienta `Bison` que se emplea en las prácticas hace uso de este método.

primeros componentes de los ítems correspondientes. Sucede con los estados I_3 e I_6, los estados I_4 e I_7, y los estados I_8 e I_9.

Esta coincidencia tiene una explicación sencilla: si hubiésemos construido la colección $LR(0)$, por cada uno de estos pares de estados existiría un único estado, ya que la primera parte es exactamente igual en los ítems $LR(0)$ y los $LR(1)$. Recuérdese que la diferencia entre ambos tipos de ítems son los caracteres de anticipación.

El método $LALR$ se basa en esta idea para reducir el número de conjuntos de ítems de la colección $LR(1)$ antes de construir la tabla. Si, por ejemplo, fusionamos los estados I_4 e I_7 del ejemplo 3.18 en un único estado, formaríamos el conjunto de ítems $I_{4,7} = \{\ [C \to d\cdot, c/d/\$]\ \}$. Obsérvese que al unir los ítems se unen también los símbolos de anticipación de la segunda parte de los mismos.

De forma similar, las transiciones de salida o entrada de un estado de la colección $LR(1)$ que se fusiona con otros, pasarán a ser transiciones de salida o entrada del estado fusionado. Por ejemplo, la transición desde I_0 a I_4 del ejemplo 3.18 se convierte en una transición desde I_0 a $I_{4,7}$.

La colección de ítems construida de esta forma se denomina colección $LALR$, y si la tabla resultante no contiene conflictos, la gramática es $LALR$. La cantidad de estados obtenidos es exactamente igual a la que se conseguiría con el método SLR. El siguiente algoritmo formaliza la construcción de tablas $LALR$.

Algoritmo 3.11. Construcción de una tabla de análisis sintáctico $LALR$.
Entrada: una gramática libre de contexto aumentada G'.
Salida: las tablas ACCIÓN e IR-A para el análisis $LALR$ de G'.

---

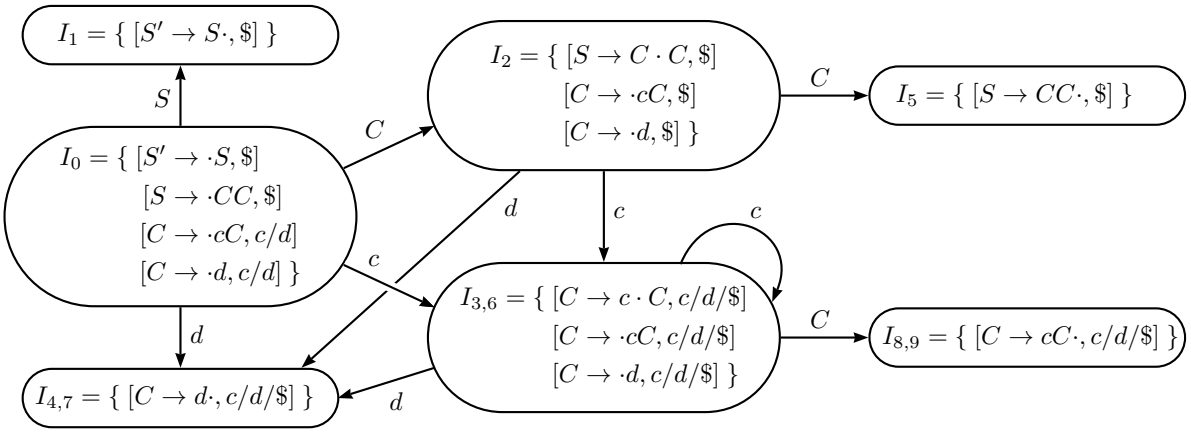
1. Constrúyase la colección $LR(1)$ $C = \{I_0, I_1, \ldots, I_n\}$ para G', usando el algoritmo 3.9;

2. Encuéntrense los conjuntos de ítems en los que los primeros componentes de todos sus items coinciden, y únanse en un único estado que sustituya a los conjuntos coincidentes.

3. Sea $C' = \{J_0, J_1, \ldots, J_m\}$ la nueva colección de ítems obtenida. Construir las acciones para el estado i a partir del conjunto J_i de la forma indicada en el algoritmo 3.10. Si no existe ningún conflicto en las acciones, se dice que la gramática es $LALR(1)$. En caso contrario, el algoritmo no consigue producir un analizador sintáctico.

4. Las transiciones IR-A para el estado i se determinan de la siguiente forma:
   Si $J_i = I_1 \cup I_2 \cup \ldots I_k$, entonces GOTO$(I_1, X)$, GOTO$(I_2, X)$, ..., GOTO$(I_k, X)$ son estados de $LR(1)$ que coinciden en el conjunto formado por el primer componente de todos sus ítems respectivos. Sea J_t el estado resultante de la fusión de estos estados en el paso 3. Entonces definimos GOTO$(J_i, X) = J_t$.

Ejemplo 3.20. La siguiente figura muestra la colección de ítems $LALR$ obtenida en el paso 2 del algoritmo anterior:



La tabla de análisis obtenida con la aplicación de los pasos 3 y 4 del algoritmo es la siguiente:

ESTADO	ACCIÓN			IR-A	
	c	d	\$	S	C
0	d36	d47		1	2
1			acc		
2	d36	d47			5
36	d36	d47			89
47	r3	r3	r3		
5			r1		
89	r2	r2	r2		

Es interesante tener en cuenta que los símbolos de anticipación de los ítems de la colección $LALR$ son subconjuntos de los conjuntos SIGUIENTE de los no terminales correspondientes, de modo que las acciones de reducción se pueden aplicar de modo más preciso que con el método SLR. Por ello, pueden existir gramáticas que sean $LALR$ pero no SLR. Es el caso de la gramática del ejercicio 3.16.

En el caso en que los símbolos de anticipación de los ítems $LALR$ que producen acciones de reducción coincidan exactamente con los conjuntos SIGUIENTE de los no terminales en la cabeza de dichos ítems, se llega a una situación interesante, ya que la tabla $LALR$ coincide con la tabla SLR.

3.5.6.1. Conflictos en las tablas LALR

En esta sección vamos a comprobar que, si una gramática es LR-canónica, el método $LALR$ puede generar conflictos del tipo *reduce/reduce*, pero no puede generar conflictos del tipo *desplaza/reduce*.

Supongamos los siguientes conjuntos de ítems en la colección $LR(1)$:

- $I_i : \{ \ldots, [A \to \alpha\cdot, a], [B \to \beta\cdot, b], \ldots \}$
- $I_j : \{ \ldots, [A \to \alpha\cdot, c], [B \to \beta\cdot, a], \ldots \}$

Si partimos de la hipótesis de que la gramática es LR-canónica, estos conjuntos no deben presentar ningún conflicto. Por tanto, $a \neq b$ y $c \neq a$. No obstante, al unificar los conjuntos I_i e I_j obtenemos:

$$I_{ij} : \{ \ldots, [A \to \alpha\cdot, a/c], [B \to \beta\cdot, b/a], \ldots \}$$

dando lugar a un conflicto del tipo *reduce/reduce*, ya que la tabla de análisis tendrá las acciones reduce $A \to \alpha$ y reduce $B \to \beta$ en la entrada ACCIÓN$[ij, a]$. Por tanto, es posible la aparición de conflictos *reduce/reduce* en la tabla $LALR$, partiendo de una gramática LR-canónica.

Veamos ahora el otro tipo de conflicto. Tomando como hipótesis, de nuevo, que la gramática es LR-canónica, para que exista un conflicto *desplaza/reduce* en la tabla $LALR$ debería obtenerse en algún conjunto de la colección $LALR$, como $J_k = I_i \cup I_j \cup \ldots$, un par de ítems del tipo:

$$J_k : \{ \ldots, [A \to \alpha \cdot a\beta, x/y/\ldots], [B \to \beta\cdot, a/z/\ldots], \ldots \}$$

Pero para que esto sea posible, debe existir un par de ítems $[A \to \alpha \cdot a\beta, x]$ y $[B \to \beta\cdot, a]$ previamente en algún conjunto de ítems $LR(1)$, como I_i o I_j, de modo que la gramática tampoco sería LR-canónica, ya que el conflicto *desplaza/reduce* existiría en uno de los dos conjuntos, contradiciendo la hipótesis de partida. Por tanto, no pueden aparecer conflictos *desplaza/reduce* en la tabla $LALR$ si la gramática es LR-canónica.

El lector puede comprobar que la siguiente gramática es LR-canónica, pero no $LALR$:

$$\begin{aligned} S &\to aAd \mid bBd \mid aBe \mid bAe \\ A &\to c \\ B &\to c \end{aligned}$$

3.5.7. Ambigüedad en el análisis LR

Ninguna gramática ambigua puede ser $LR(1)$ ni $LL(1)$. En la sección 3.7.2.3 se tratará el problema de la ambigüedad en las gramáticas $LL(1)$. Veremos que para resolver la ambigüedad en este caso, se requiere un ajuste manual de la tabla de análisis para eliminar los conflictos y seleccionar las reglas que corrigen la ambigüedad.

En esta sección se va a mostrar que sucede algo similar cuando se manipula una gramática ambigua con el análisis LR. Para ello, se emplea la gramática de sentencias `if-then` e `if-then-else`:

$$\begin{aligned} sent &\to \textbf{if } expr \textbf{ then } sent \\ &\mid \textbf{if } expr \textbf{ then } sent \textbf{ else } sent \\ &\mid \textbf{s} \\ expr &\to \textbf{e} \end{aligned}$$

En la sección 3.2.2.2 se obtuvo la siguiente gramática no ambigua equivalente:

$$\begin{aligned} sent &\to sent_e \\ &\mid sent_{ne} \\ sent_e &\to \textbf{if } expr \textbf{ then } sent_e \textbf{ else } sent_e \\ &\mid \textbf{s} \\ sent_{ne} &\to \textbf{if } expr \textbf{ then } sent \\ &\mid \textbf{if } expr \textbf{ then } sent_e \textbf{ else } sent_{ne} \\ expr &\to \textbf{e} \end{aligned}$$

Demostraremos en la sección 3.7.2.3 que esta gramática no es $LL(1)$. No obstante, el lector puede comprobar que sí es LR, si bien se obtiene una tabla de análisis es excesivamente extensa.

Una solución alternativa es construir la tabla de análisis de la gramática ambigua original, y resolver los conflictos que aparezcan debidos a la ambigüedad, usando el criterio conocido de asociar el `else` al `if` más cercano.

Puesto que el problema de la ambigüedad se debe a las reglas del no terminal $sent$ y, más concretamente, al token **else**, se puede emplear la siguiente gramática simplificada:

$$\begin{aligned} S &\to \textbf{i } S \textbf{ e } S \\ &\mid \textbf{i } S \\ &\mid \textbf{s} \end{aligned}$$

Obsérvese que se ha eliminado el no terminal $expr$, el terminal **then** y se han empleado símbolos de una sola letra para simplificar la construcción de la colección de ítems ($S = sent$, **i** = **if** y **e** = **else**). La colección de ítems $LR(0)$ es:

$$I_0 = \{\ S' \to \cdot S \quad S \to \cdot \mathbf{i}S\mathbf{e}S \quad S \to \cdot \mathbf{i}S \quad S \to \cdot \mathbf{s}\ \}$$

$$I_1 = \{\ S' \to S\cdot\ \}$$

$$I_2 = \{\ S \to \mathbf{i}\cdot S\mathbf{e}S \quad S \to \mathbf{i}\cdot S \quad S \to \cdot \mathbf{i}S\mathbf{e}S \quad S \to \cdot \mathbf{i}S \quad S \to \cdot \mathbf{s}\ \}$$

$$I_3 = \{\ S \to \mathbf{s}\cdot\ \}$$

$$I_4 = \{\ S \to \mathbf{i}S\cdot \mathbf{e}S \quad S \to \mathbf{i}S\cdot\ \}$$

$$I_5 = \{\ S \to \mathbf{i}S\mathbf{e}\cdot S \quad S \to \cdot \mathbf{i}S\mathbf{e}S \quad S \to \cdot \mathbf{i}S \quad S \to \cdot \mathbf{s}\ \}$$

$$I_6 = \{\ S \to \mathbf{i}S\mathbf{e}S\cdot\ \}$$

La tabla de análisis SLR obtenida a partir de la colección de ítems es la siguiente::

ESTADO	ACCIÓN				IR-A
	i	**e**	**s**	$	S
0	d2		d3		1
1				acc	
2	d2		d3		4
3		r3		r3	
4		d5/r2		r2	
5	d2		d3		6
6		r1		r1	

Para resolver el conflicto de la casilla $M[4, \mathbf{e}]$ hay que consultar el conjunto de ítems I_4 y razonar sobre cuál es la acción que permite imponer la regla de asociar el **else** al **if** más próximo. Si eligiésemos la reducción, estaríamos disociando el **else** del **if** inmediatamente anterior. Sin embargo, el desplazamiento introduce en la pila el **else** para vincularlo más adelante con el **if** anterior mediante la reducción de la regla 1. Por tanto, el desplazamiento es la acción correcta.

La siguiente simulación muestra un análisis de la entrada $w =$ **iises**\$[8] que resuelve el conflicto de la forma indicada:

PILA	ENTRADA	ACCIÓN
0	**iises**$	d2
0 **i** 2	**ises**$	d2
0 **i** 2 **i** 2	**ses**$	d3
0 **i** 2 **i** 2 **s** 3	**es**$	r3 $S \to \mathbf{s}$
0 **i** 2 **i** 2 S 4	**es**$	d5
0 **i** 2 **i** 2 S 4 **e** 5	**s**$	d3
0 **i** 2 **i** 2 S 4 **e** 5 **s** 3	$	r3 $S \to \mathbf{s}$
0 **i** 2 **i** 2 S 4 **e** 5 S 6	$	r1 $S \to \mathbf{i}S\mathbf{e}S$
0 **i** 2 S 4	$	r2 $S \to \mathbf{i}S$
0 S 1	$	acc

La derivación obtenida es: $S \Rightarrow \mathbf{i}S \Rightarrow \mathbf{ii}S\mathbf{e}S \Rightarrow \mathbf{ii}S\mathbf{es} \Rightarrow \mathbf{iises}$.

En conclusión, es posible el análisis de gramáticas ambiguas usando el método LR. Si bien la gramática ambigua no puede ser LR-canónica, y por extensión tampoco $LALR$ ni SLR, los conflictos de las tablas de análisis se pueden resolver de forma manual, siguiendo criterios razonados.

[8] Esta entrada representa una sentencia del tipo: `if (x1) if (x2) then s1 else s2;`.

3.5.8. Recuperación de errores en el análisis LR

Si observamos el algoritmo 3.4, y las distintas tablas de análisis generadas según las técnicas SLR, LR-canónica y $LALR$ mediante los algoritmos 3.7, 3.10 y 3.11 respectivamente, veremos que los errores en el análisis sintáctico sólo se van a detectar cuando se acceda a la parte de ACCIÓN de la correspondiente tabla de análisis sintáctico, y se encuentre una celda vacía.

La parte de IR-A para los no terminales de la gramática nunca producirá errores. Dicho de otra forma, nunca se accederá a una celda vacía en las columnas correspondientes a los símbolos no terminales. Esto es así porque cuando se aplica una reducción mediante una producción, tenemos garantía de que el nuevo conjunto de símbolos de la pila forman un prefijo viable.

Cualquiera de los métodos LR es capaz de detectar un error antes de meter un símbolo erróneo en la pila. Con el método LR-canónico, el error se detecta inmediatamente. Con los métodos $LALR$ o SLR pueden hacerse varias reducciones antes de detectar el error en la entrada, pero nunca desplazarán un símbolo de entrada erróneo a la pila.

3.5.8.1. Tratamiento a nivel de frase

En la recuperación de errores a nivel de frase, se modifica la secuencia de tokens que se reciben desde la entrada para intentar convertirla en legal.

Para esto resulta útil basarse en los errores cometidos por el programador más frecuentemente; además, hay que tener en cuenta las particularidades del lenguaje de programación.

Una buena estrategia para llevar a cabo este tipo de recuperación puede ser la siguiente: para cada entrada en blanco de la tabla de análisis, introducir una rutina de manejo del error que realice la acción apropiada para el estado en el que estamos, y el símbolo que se está leyendo. Dentro de las posibles acciones que se pueden realizar, tenemos:

- Inserción o eliminación de símbolos en la pila.
- Inserción o eliminación de símbolos en la entrada.
- Alteración o transposición de símbolos en la entrada.

Debe evitarse, al extraer símbolos de la pila, sacar de la misma un símbolo no terminal ya que su presencia en ella indica que se había reconocido una estructura con éxito.

Ejemplo 3.21. En la tabla construida en el ejemplo 3.15, podemos insertar una rutina de error en las entradas vacías de los estados 0, 4, 6 y 7. En estos estados se espera un operando, es decir, un token del tipo id o (. Si no se detecta uno de estos dos tokens, la rutina de error se puede encargar de introducir un id imaginario en la pila y, seguidamente, el estado 3. Además, se emite el mensaje de error que indica la falta de un operando.

3.5.8.2. Tratamiento de errores en modo pánico

En modo pánico, la situación es ahora diferente a la que se presenta en el análisis descendente (véase 3.7.4.1). En aquel caso, se conoce cuál es el símbolo no terminal que se tiene que derivar y se necesita determinar si el símbolo de anticipación pertenece al conjunto Predict de una producción o al conjunto SIGUIENTE del no terminal.

Con el análisis ascendente la situación es distinta; disponemos de una parte derecha incompleta y tenemos que encontrar un símbolo no terminal que sea adecuado, para aplicar una reducción y sustituirla por éste. Por lo tanto, es relavitamente arbitrario el símbolo no terminal a elegir para simular una reducción.

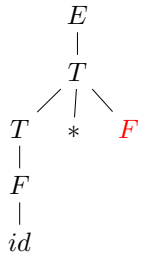
Para seguir un procedimiento sistemático, una vez detectado un error, se aplican los siguientes pasos:

1. Se extraen símbolos de la pila hasta encontrar un estado s que tenga un valor IR-A$(s, A) = s'$ definido en la tabla de análisis. Pueden existir varios no terminales A que cumplan esta condición. Para intentar limitar la propagación del error, se puede seleccionar el no terminal que esté a mayor nivel de profundidad en la jerarquía que determinan las reglas de producción de la gramática.
2. Se introduce A en la pila, y seguidamente s'.
3. Se ignoran símbolos de la entrada hasta que se encuentre uno considerado de sincronización para A. Se tratará de un símbolo que pertenezca al conjunto SIGUIENTE(A).
4. A partir de aquí se sigue el análisis normalmente.

Ejemplo 3.22. Retomamos la tabla construida en el ejemplo 3.15, para aplicar el algoritmo de análisis LR a la cadena de entrada errónea $w = id * id($, con la recuperación de errores en modo pánico. A continuación se muestran los pasos de la simulación:

PILA	ENTRADA	ACCIÓN
0	$id * id$ (\$	d5
0 id 5	$* id$ (\$	r6 $F \to id$
0 F 3	$* id$ (\$	r4 $T \to F$
0 T 2	$* id$ (\$	d7
0 T 2 $*$ 7	id (\$	d5
0 T 2 $*$ 7 id 5	(\$	**Error**: eliminar id 5 de la pila, insertar F 10 en la pila, y eliminar (de la entrada.
0 T 2 $*$ 7 F 10	\$	r3 $T \to T * F$
0 T 2	\$	r2 $E \to T$
0 E 1	\$	acc

Obviamente, el tratamiento de errores introduce nodos ficticios en el árbol de análisis, de modo que no es posible la generación de código en etapas posteriores. El árbol obtenido con la simulación anterior es:



3.6. Algoritmos de transformación de gramáticas

En esta sección se tratan algunos algoritmos de transformación de las gramáticas libres de contexto que pueden ser necesarios para poder aplicar algunos de los métodos de análisis.

En concreto, para el uso del método de análisis descendente descrito en la sección 3.7, los algoritmos de transformación mostrados a continuación se deben aplicar en cadena, siguiendo el orden indicado en la figura 3.5. No será necesario aplicar siempre todos algoritmos, ya que la gramática puede cumplir las características buscadas antes de aplicar alguna de las transformaciones.

No se incluye en esta sección ningún algoritmo para la eliminación de la ambigüedad, ya que como se indicó en el apartado 3.2.2, no hay un procedimiento sistemático para su tratamiento. Ahora bien, los métodos estudiados en las secciones 3.7 y 3.5 no se pueden aplicar a gramáticas ambiguas, a no ser que se empleen soluciones específicas para cada caso.

3.6.1. Eliminación de λ-producciones

Definición 3.25. *Se dice que una gramática libre de contexto $G = (V_N, V_T, S, P)$ es* λ-libre *si no contiene producciones de la forma $A \to \lambda$, excepto a lo sumo $S \to \lambda$, con la condición de que S no aparezca en la parte derecha de ninguna regla de producción.*

Teorema 3.1. *Dada una gramática libre de contexto $G = (V_N, V_T, S, P)$, existe otra gramática libre de contexto equivalente $G' = (V'_N, V_T, S', P')$ que es λ-libre.*

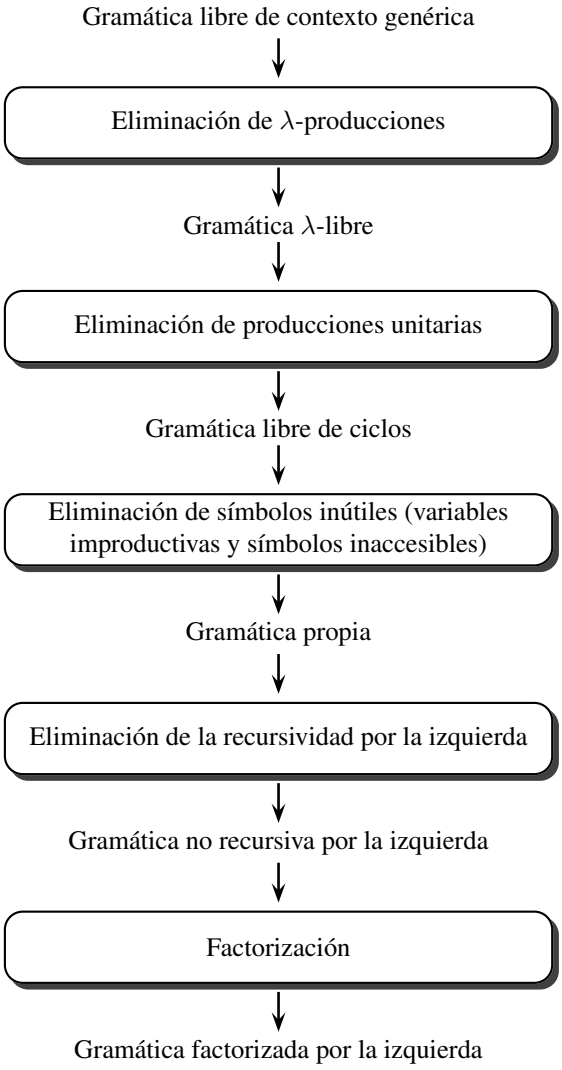


Figura 3.5: Algoritmos de transformación de gramáticas.

Algoritmo 3.12. Eliminación de λ-producciones.
Entrada: una gramática libre de contexto $G = (V_N, V_T, S, P)$.
Salida: una gramática equivalente λ-libre $G' = (V'_N, V_T, S', P')$.

---

1. **Obtener el conjunto de variables anulables** $V_{anu} = \{A \in V_N | A \Rightarrow^* \lambda\}$ (derivan en λ):

   1.1. Inicializar V_{anu} añadiendo todas las variables A tal que $A \to \lambda \in P$;

   1.2. REPETIR:

   Por cada regla del tipo $B \to C_1 C_2 \dots C_n$, donde todas las variables $C_i \in V_{anu}$ (son anulables) entonces se añade B a V_{anu};

   HASTA que no se añadan variables nuevas a V_{anu};

2. **Eliminar λ-reglas y añadir nuevas**

   2.1. Eliminar de P todas las reglas de la forma $A \to \lambda$;

   2.2. Por cada regla $A \to \alpha \in P$ se añaden a P todas las reglas que se generan al considerar que cada variable anulable que aparece en α se incluye en una regla y no se incluye en otra regla (si hay

n variables anulables en α entonces se podrían generar 2^n reglas);

2.3. Eliminar de P las λ-reglas que se hayan generado en el paso anterior (ocurre cuando toda la parte derecha de una regla contiene sólo anulables);

3. **Añadir nuevo estado inicial y reglas adicionales si es necesario**

3.1. Si S está en V_{anu} ($S \Rightarrow^* \lambda$ en la gramática de entrada) entonces:

3.1.1. Si S no aparece en la parte derecha entonces añadir la regla $S \to \lambda$ a P;

3.1.2. En otro caso

Considerar un **nuevo símbolo inicial** S' y añadir S' a V_N;
Añadir las reglas $S' \to S \mid \lambda$ a P;

4. Devolver (G'), con los componentes de G modificados por los pasos anteriores.

Al finalizar la aplicación del algoritmo, G' podría quedar con símbolos inútiles.

3.6.2. Eliminación de producciones unitarias

Definición 3.26. *Llamamos producciones unitarias a las que tienen la forma $A \to B$, con $A, B \in V_N$.*

Teorema 3.2. *Dada una gramática libre de contexto $G = (V_N, V_T, S, P)$, existe otra gramática libre de contexto equivalente $G' = (V_N, V_T, S, P')$ que no contiene producciones unitarias.*

Algoritmo 3.13. Eliminación de producciones unitarias.
Entrada: una gramática λ-libre $G = (V_N, V_T, S, P)$.
Salida: una gramática equivalente sin producciones unitarias $G' = (V_N, V_T, S, P')$.

---

1. Para cada variable $A \in V_N$ se calcula $V_{uni}(A) = \{B \in V_N | A \Rightarrow^* B, B \neq A\}$, que es el conjunto de **variables que se derivan de A por reglas unitarias**;

2. **Eliminar las reglas unitarias** de P;

3. **Añadir nuevas reglas:** para cada variable A tal que $V_{uni}(A) \neq \varnothing$ hacer
   Para cada variable $B \in V_{uni}(A)$
   Para cada regla de la forma $B \to \beta$
   Añadir la regla $A \to \beta$ a P;

4. Eliminar de V_N todas aquellas variables que no aparecen en las reglas que quedan en P;

5. Devolver (G'), con los componentes de G modificados por los pasos anteriores.

La eliminación de las producciones unitarias de la gramática garantiza la eliminación de ciclos $A \Rightarrow^+ A$. Al igual que en el algoritmo anterior, la gramática de salida G' puede contener símbolos inútiles.

3.6.3. Eliminación de símbolos inútiles

Definición 3.27. *Sea $G = (V_N, V_T, S, P)$ una gramática libre de contexto. Decimos que un símbolo de la gramática $X \in V_N \cup V_T$ es útil si existe una derivación de la forma:*

$$S \Rightarrow^* \alpha X \beta \Rightarrow^* w$$

donde $w \in V_T^*$ y $\alpha, \beta \in (V_N \cup V_T)^*$.

Teorema 3.3. *Dada una gramática libre de contexto $G = (V_N, V_T, S, P)$, con $L(G) \neq \emptyset$, existe otra gramática libre de contexto equivalente $G' = (V_N', V_T', S, P')$ sin símbolos inútiles.*

La eliminación de los símbolos inútiles de una gramática requiere la aplicación de dos algoritmos[9]:

1. Eliminación de *variables improductivas*.
2. Eliminación de *símbolos inaccesibles*.

3.6.3.1. Eliminación de variables improductivas

Definición 3.28. *Una variable $A \in V_N$ es improductiva si no existen derivaciones $A \Rightarrow^* w$, $w \in V_T^*$.*

Teorema 3.4. *Dada una gramática libre de contexto $G = (V_N, V_T, S, P)$, con $L(G) \neq \emptyset$, existe otra gramática libre de contexto equivalente $G' = (V_N', V_T, S, P')$ sin variables improductivas.*

Algoritmo 3.14. Eliminación de variables improductivas.
Entrada: una gramatica libre de contexto $G = (V_N, V_T, S, P)$.
Salida: una gramática equivalente sin variables improductivas $G' = (V_N', V_T, S, P')$.

---

1. Inicializar un conjunto V_{pro} de variables productivas añadiendo todas las variables A de V_N para las que existe una regla $A \rightarrow w$, tal que $w \in V_T^*$ (sólo deriva en terminales);
2. REPETIR:

   Examinar cada regla $B \rightarrow \alpha$ de P y si todas las variables que aparecen en α están ya en V_{pro} entonces añadir B a V_{pro};

   HASTA que no se añadan variables nuevas a V_{pro};
3. Hacer $V_N \leftarrow V_{pro}$ y de esa forma se eliminan las variables improductivas de V_N;
4. Eliminar de P aquellas reglas que tienen alguna variable improductiva (no pertenece a V_N) en la parte izquierda o derecha;
5. Devolver (G'), con los componentes de G modificados por los pasos anteriores.

3.6.3.2. Eliminación de símbolos inaccesibles

Definición 3.29. *Un símbolo $X \in (V_N \cup V_T)$ es inaccesible si no aparece en ninguna forma sentencial de la gramática, es decir, $\nexists \alpha, \beta \in (V_N \cup V_T)^*$ tal que $S \Rightarrow^* \alpha X \beta$.*

[9] El orden de aplicación de los algoritmos debe ser el indicado.

Teorema 3.5. *Dada una gramática libre de contexto $G = (V_N, V_T, S, P)$, con $L(G) \neq \emptyset$, existe otra gramática libre de contexto equivalente $G' = (V'_N, V_T, S, P')$ sin símbolos inaccesibles.*

Algoritmo 3.15. Eliminación de símbolos inaccesibles.
Entrada: una gramatica libre de contexto $G = (V_N, V_T, S, P)$.
Salida: una gramática equivalente sin símbolos inaccesibles $G' = (V'_N, V'_T, S, P')$.

---

1. Inicializar un conjunto V_{acc} de variables accesibles añadiendo el símbolo inicial S;
2. REPETIR:

   Para cada variable A en V_{acc} examinar cada regla del tipo $A \to \alpha$ en P y añadir a V_{acc} todas las variables que aparecen en α;

   HASTA que no se añadan variables nuevas a V_{acc};
3. Hacer $V_N \leftarrow V_{acc}$ y de esa forma se eliminan las variables inaccesibles de V_N;
4. Eliminar de P aquellas reglas que tienen alguna variable inaccesible (ya no pertenece a V_N) en la parte izquierda o derecha;
5. Eliminar de V_T todos aquellos símbolos terminales que no aparecen en las reglas que quedan en P (son terminales inaccesibles);
6. Devolver (G'), con los componentes V_N, V_T y P modificados por los pasos anteriores.

3.6.4. Gramáticas propias

Definición 3.30. *Una gramática es propia si es λ-libre, libre de ciclos y no tiene símbolos inútiles.*

Teorema 3.6. *Dada una gramática libre de contexto $G = (V_N, V_T, S, P)$, con $L(G) \neq \emptyset$, existe otra gramática libre de contexto equivalente $G' = (V'_N, V'_T, S', P')$ que es propia.*

Para convertir una gramática en otra equivalente propia, podemos seguir los siguientes pasos:

1. Transformar la gramática en una equivalente λ-libre.
2. Eliminar las producciones unitarias (se eliminan los ciclos en el caso de que los hubiese).
3. Eliminar los símbolos inútiles.

Es importante indicar que una gramática puede tener producciones unitarias y ser propia.

3.6.5. Eliminación de la recursividad por la izquierda

Definición 3.31. *Una gramática libre de contexto $G = (V_N, V_T, S, P)$ se dice que es:*

- recursiva por la izquierda *si $\exists A \in V_N \mid A \Rightarrow^+ A\alpha$. En este caso A es una* variable recursiva por la izquierda.
- recursiva por la derecha *si $\exists A \in V_N \mid A \Rightarrow^+ \alpha A$. En este caso A es una* variable recursiva por la derecha.
- recursiva *si $\exists A \in V_N \mid A \Rightarrow^+ \alpha A \beta$.*

Definición 3.32. *Una regla de producción es:*

- recursiva por la izquierda *si es de la forma* $A \to A\alpha$.
- recursiva por la derecha *si es de la forma* $A \to \alpha A$.
- recursiva *si es de la forma* $A \to \alpha A \beta$.

Teorema 3.7. *Dada una gramática libre de contexto* $G = (V_N, V_T, S, P)$ *recursiva por la izquierda, existe otra gramática libre de contexto equivalente* $G' = (V'_N, V_T, S, P')$ *que no es recursiva por la izquierda.*

Los métodos de análisis sintáctico descendente que construyen derivaciones más a la izquierda no pueden manejar gramáticas recursivas por la izquierda. El método para eliminar este tipo de recursividad se basa en dos algoritmos:

1. Eliminación de la recursividad inmediata por la izquierda.
2. Eliminación de la recursividad por la izquierda de toda la gramática.

3.6.5.1. Eliminación de la recursividad inmediata por la izquierda

Algoritmo 3.16. Eliminación de la recursividad inmediata por la izquierda de una variable A.
Entrada: una gramática λ-libre $G = (V_N, V_T, S, P)$ con una variable A recursiva por la izquierda.
Salida: una gramática equivalente $G' = (V'_N, V_T, S, P')$ sin recursividad izquierda en A.

---

1. Ordenar las producciones de A:
   $A \to A\alpha_1 \mid A\alpha_2 \mid \ldots \mid A\alpha_m \mid \beta_1 \mid \beta_2 \mid \ldots \mid \beta_n$;
2. Añadir una variable nueva A' a V_N;
3. Reemplazar en P las A-producciones por las siguientes nuevas reglas de producción:
   $A \to \beta_1 \mid \beta_2 \mid \ldots \mid \beta_n \mid \beta_1 A' \mid \beta_2 A' \mid \ldots \mid \beta_n A'$;
   $A' \to \alpha_1 \mid \alpha_2 \mid \ldots \mid \alpha_m \mid \alpha_1 A' \mid \alpha_2 A' \mid \ldots \mid \alpha_m A'$;
4. Devolver (G'), con los componentes de G modificados por los pasos anteriores.

Es importante tener en cuenta que la eliminación de la recursividad inmediata por la izquierda no implica que la gramática resultante no sea recursiva por la izquierda, puesto que puede ocurrir que $A \Rightarrow^+ A\alpha$.

Ejemplo 3.23. La siguiente gramática:

$$E \to E + T \mid T$$
$$T \to T * F \mid F$$
$$F \to (E) \mid id$$

es recursiva por la izquierda, puesto que tiene dos variables (E y T) que presentan recursividad inmediata por la izquierda. La gramática es λ-libre, de modo que podemos aplicar el algoritmo expuesto anteriormente para eliminar la recursividad de ambas variables.

Dadas las reglas de producción de E, es fácil determinar qué valor toman α_1 y β_1:

$$E \to E \underbrace{+T}_{\alpha_1} \mid \underbrace{T}_{\beta_1}$$

Por tanto, el algoritmo nos indica que debemos sustituir esas reglas de producción de E por estas otras:

$$E \to T \mid TE'$$
$$E' \to +T \mid +TE'$$

Aplicando un razonamiento similar para las reglas de producción de T, obtenemos las nuevas reglas:

$$T \to F \mid FT'$$
$$T' \to *F \mid *FT'$$

3.6.5.2. Eliminación de la recursividad por la izquierda de toda la gramática

El algoritmo anterior elimina la recursividad de un paso de derivación, pero no la que puede aparecer en dos o más pasos. Para eliminar la recursividad de este tipo, en el caso de que la gramática la presentase, es necesario aplicar el siguiente algoritmo.

Algoritmo 3.17. Eliminación de la recursividad por la izquierda de toda la gramática.
Entrada: una gramática $G = (V_N, V_T, S, P)$ propia.
Salida: una gramática equivalente $G' = (V'_N, V_T, S, P')$ sin recursividad por la izquierda.

---

1. Ordenar los no terminales $V_N = \{A_1, A_2, \ldots, A_n\}$, donde $A_1 = S$.
2. Inicializar $V'_N \leftarrow V_N$ y $P' \leftarrow P$;
3. REPETIR desde $i = 1$ HASTA n:
   - 3.1. Si hay recursión inmediata en las reglas para A_i entonces:
     - 3.1.1 $V'_N \leftarrow V'_N \cup \{A'_i\}$;
     - 3.1.2 Sustituir $A_i \to A_i\alpha_1 \mid A_i\alpha_2 \mid \ldots \mid A_i\alpha_m \mid \beta_1 \mid \beta_2 \mid \ldots \mid \beta_n$ en P' por:
       $A_i \to \beta_1 \mid \beta_2 \mid \ldots \mid \beta_n \mid \beta_1 A'_i \mid \beta_2 A'_i \mid \ldots \mid \beta_n A'_i$;
       $A'_i \to \alpha_1 \mid \alpha_2 \mid \ldots \mid \alpha_m \mid \alpha_1 A'_i \mid \alpha_2 A'_i \mid \ldots \mid \alpha_m A'_i$;
   - 3.2. Si $i < n$ entonces REPETIR desde $j = 1$ HASTA i:
     - 3.2.1. Sustituir cada producción $A_{i+1} \to A_j\alpha$ en P' por:
       $A_{i+1} \to \gamma_1\alpha \mid \gamma_2\alpha \mid \ldots \mid \gamma_p\alpha$;
       donde $A_j \to \gamma_1 \mid \gamma_2 \mid \ldots \mid \gamma_p$.
4. Devolver (G'), con los componentes de V'_N y P' modificados por los pasos anteriores.

El algoritmo requiere una gramática de entrada que sea propia. En caso de que no lo sea, será necesario aplicar las transformaciones indicadas anteriormente.

La eliminación de la recursividad izquierda se apoya en una ordenación de los no terminales de la gramática de entrada. La iteración principal recorre los no terminales en este orden, y por cada uno de ellos comprueba si existe recursividad izquierda inmediata. En caso de que así sea, se aplica el algoritmo de eliminación de la recursividad inmediata para la variable actual A_i. Seguidamente, el algoritmo comprueba si alguna de las reglas de producción de la siguiente variable, A_{i+1}, tiene al comienzo de su parte derecha alguno de los no terminales anteriores en la ordenación establecida en el primer paso del algoritmo, es decir, desde A_1 hasta A_i. Para eliminar esta situación, que puede conducir a una recursividad izquierda en dos o más pasos, se modifican las reglas de producción de A_{i+1}, sustituyendo los no terminales A_1 hasta A_i que aparezcan al comienzo del lado derecho por el cuerpo de sus respectivas reglas de producción.

Ejemplo 3.24. Continuando con el ejemplo 3.23, podemos aplicar el algoritmo de eliminación de la recursividad en toda la gramática. La gramática es propia aún teniendo producciones unitarias, de modo que el algoritmo es aplicable sin necesidad de realizar transformaciones previas. Usando la ordenación $\{E, T, F\}$ el algoritmo realiza los siguientes pasos:

- $i = 1$. Se elimina la recursividad inmediata de las reglas de E del modo indicado en el ejemplo 3.23. La nueva gramática en este punto es:

$$\begin{aligned} E &\to T \mid TE' \\ E' &\to +T \mid +TE' \\ T &\to T * F \mid F \\ F &\to (E) \mid id \end{aligned}$$

  Podemos comprobar que no existe ninguna regla del tipo $T \to E\alpha$, de modo que no se aplica el punto 3.2.1. del algoritmo.

- $i = 2$. Se elimina la recursividad inmediata de las reglas de T. La nueva gramática en este punto es:

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

  Respecto al punto 3.2.1, comprobamos que no existen reglas del tipo $F \to E\alpha$ ni $F \to T\alpha$.

- $i = 3$. F no presenta recursividad inmediata por la izquierda. La gramática final es la obtenida en la iteración anterior.

Ejemplo 3.25. Veamos otro ejemplo de aplicación del algoritmo 3.17. Consideramos la siguiente gramática libre de contexto $G = (V_N, V_T, S, P)$:

$$\begin{aligned} S &\to AA \mid 0 \\ A &\to SS \mid 1 \end{aligned}$$

La gramática es recursiva por la izquierda indirectamente, puesto que $S \Rightarrow AA \Rightarrow SSA$. Por otra parte, la gramática es propia puesto que es λ-libre, libre de ciclos y no tiene símbolos inútiles. Por tanto, es aplicable directamente el algoritmo 3.17. Usando la ordenación $\{S, A\}$, se realizan los siguientes pasos:

- $i = 1$. No hay recursividad inmediata en las reglas de S.
  - $j = 1$. Se sustituye la regla $A \to SS$ por las dos nuevas reglas $A \to AAS \mid 0S$, de modo que la gramática se transforma de la siguiente forma:

$$\begin{aligned} S &\to AA \mid 0 \\ A &\to AAS \mid 0S \mid 1 \end{aligned}$$

- $i = 2$. Hay recursividad directa con A:

$$A \to A\underbrace{AS}_{\alpha_1} \mid \underbrace{0S}_{\beta_1} \mid \underbrace{1}_{\beta_2}$$

  La gramática se transforma de la siguiente forma:

$$\begin{aligned} S &\to AA \mid 0 \\ A &\to 0S \mid 1 \mid 0SA' \mid 1A' \\ A' &\to AS \mid ASA' \end{aligned}$$

La gramática resultante es la obtenida en el último paso indicado.

3.6.6. Factorización

Supongamos que una gramática $G = (V_N, V_T, S, P)$ tiene varias reglas de producción de un no terminal con un prefijo común en su lado derecho, es decir, $A \to \alpha\beta_1 \mid \alpha\beta_2 \mid \ldots \mid \gamma$, donde γ representa a todas las partes derechas que no comienzan con α. Esta situación plantea un problema para los métodos de análisis descendente, ya que no es inmediato determinar qué alternativa hay que utilizar para derivar el no terminal A, puesto que varias de ellas comienzan con la misma cadena de símbolos α. En estos casos, se debe transformar la gramática para factorizar estas reglas de producción, empleando el algoritmo siguiente.

Teorema 3.8. *Dada una gramática libre de contexto $G = (V_N, V_T, S, P)$ con reglas de producción con factores comunes, existe otra gramática libre de contexto equivalente $G' = (V'_N, V_T, S, P')$ que no tiene factores comunes en sus reglas de producción.*

Algoritmo 3.18. Factorización por la izquierda de una gramática libre de contexto.
Entrada: una gramática libre de contexto $G = (V_N, V_T, S, P)$.
Salida: una gramática equivalente $G' = (V'_N, V_T, S, P')$ sin factores comunes por la izquierda en las reglas de producción.

---

1. REPETIR por cada no terminal $A \in V_N$

   1.1. Encontrar el prefijo α más largo común a dos o más de sus alternativas:
   $A \to \alpha\beta_1 \mid \alpha\beta_2 \mid \ldots \mid \alpha\beta_n \mid \gamma_1 \mid \gamma_2 \mid \ldots \mid \gamma_k$;
   donde cada γ_i $(1 \leq i \leq k)$ representa una alternativa que no comienza con α.

   1.2. Si $\alpha \neq \lambda$:

      1.2.1. Añadir un nuevo no terminal A' a V_N;

      1.2.2. Sustituir en P las A-producciones por estas otras:
      $A \to \alpha A' \mid \gamma_1 \mid \gamma_2 \mid \ldots \mid \gamma_k$;
      $A' \to \beta_1 \mid \beta_2 \mid \ldots \mid \beta_n$;

   HASTA que no haya dos o más alternativas para A con un prefijo común $\alpha \neq \lambda$.

2. Devolver (G'), con los componentes de G modificados por los pasos anteriores.

Este algoritmo recorre las reglas de producción de cada variable de la gramática. Si encuentra una variable con un factor común α en varias partes derechas de sus reglas de producción, sustituye las reglas con factor común por una única regla que las agrupa a todas, $A \to \alpha A'$. El nuevo no terminal A' tiene reglas de producción con las que se pueden derivar las secuencias de símbolos que quedan a continuación del factor común (β_1, β_2, etc.).

Ejemplo 3.26. En el ejemplo 3.24 se obtuvo la siguiente gramática tras aplicar el algoritmo de eliminación de la recursividad izquierda:

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

Esta gramática presenta factores comunes en las reglas de E, E', T y T'. Veamos cómo se realiza su eliminación:

- Las reglas de E son:
$$E \to \underbrace{\underbrace{T}_{\alpha}}_{\beta_1} \mid \underbrace{\underbrace{T}_{\alpha}\underbrace{E'}_{\beta_2}}_{}$$
Obsérvese que $\beta_1 = \lambda$, y que no existe ninguna alternativa γ_i sin el prefijo común. Aplicando el algoritmo de factorización, sustituimos estas producciones de E por las siguientes:
$$E \to TE''$$
$$E'' \to \lambda \mid E'$$
Obsérvese que se ha introducido el no terminal E'', para distinguirlo de E', que ya forma parte de la gramática.
- Las reglas de E' son:
$$E' \to \underbrace{\underbrace{+T}_{\alpha}}_{\beta_1} \mid \underbrace{\underbrace{+T}_{\alpha}\underbrace{E'}_{\beta_2}}_{}$$
Sustituimos estas producciones de E' por las siguientes:
$$E' \to +TE''$$
$$E'' \to \lambda \mid E'$$
Obsérvese que no es necesario introducir un nuevo no terminal para la factorización de las reglas de E', puesto que E'' puede reutilizarse.
- Las reglas de T son:
$$T \to \underbrace{\underbrace{F}_{\alpha}}_{\beta_1} \mid \underbrace{\underbrace{F}_{\alpha}\underbrace{T'}_{\beta_2}}_{}$$
Sustituimos estas producciones de T por las siguientes:
$$T \to FT''$$
$$T'' \to \lambda \mid T'$$
Se crea el nuevo no terminal T'' para distinguirlo de T'.
- Las reglas de T' son:
$$T' \to \underbrace{\underbrace{*F}_{\alpha}}_{\beta_1} \mid \underbrace{\underbrace{*F}_{\alpha}\underbrace{T'}_{\beta_2}}_{}$$
Sustituimos estas producciones de T' por las siguientes:
$$T' \to *FT''$$
$$T'' \to \lambda \mid T'$$
Como vemos, puede reutilizarse T'' para factorizar las reglas de T'.
- Las reglas de F no presentan factores comunes.

Finalmente obtenemos la siguiente gramática:

$$E \to TE''$$
$$E'' \to \lambda \mid E'$$
$$E' \to +TE''$$
$$T \to FT''$$
$$T'' \to \lambda \mid T'$$
$$T' \to *FT''$$
$$F \to (E) \mid id$$

Esta gramática puede simplificarse si eliminamos las variables E' y T', y las sustituimos por su única parte derecha en los lugares en los que aparecen:

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

Finalmente, la gramática anterior se puede expresar de forma equivalente con una notación más cómoda:

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

3.7. Análisis descendente

Los métodos de análisis sintáctico descendente usados en el desarrollo de compiladores se aplican sobre un conjunto restringido de gramáticas libres de contexto denominadas *gramáticas LL(1)*. Estas gramáticas tienen ciertas características que permiten:

- desarrollar un algoritmo de análisis de complejidad $O(n)$ en espacio y tiempo.
- analizar de forma totalmente determinista una sentencia examinando sólo una vez, de izquierda a derecha, la secuencia de tokens. A los métodos de análisis que realizan un único recorrido de la entrada se les denomina *métodos de análisis de una sola pasada*.

3.7.1. Gramáticas LL

Dada una configuración de un analizador descendente en un instante concreto, el problema clave que debe resolver el analizador es el de determinar cuál es la regla de producción que se debe aplicar a un no terminal A para conseguir derivar cierta secuencia de símbolos de la entrada. Intuitivamente se puede comprender que la complejidad lineal sólo se puede conseguir si se construye un *analizador predictivo*, es decir, un analizador que no requiera retroceso (backtracking).

Las gramáticas $LL(1)$ permiten la construcción de analizadores predictivos. En concreto, un analizador predictivo puede deducir, en cada paso de derivación, dado un terminal de la entrada a y una variable A que debe ser expandida, cuál es la alternativa de entre las A-producciones que debe ser aplicada. Es decir, si $A \to \alpha_1 \mid \alpha_2 \mid \ldots \mid \alpha_k$, sólo una de las alternativas α_i puede derivar una cadena que comience por a si queremos hacer un análisis sin retroceso. En términos del funcionamiento de los autómatas de pila (sección 3.2.3), la función de transición resulta ser determinista cuando la gramática es $LL(1)$, es decir, sólo es posible un movimiento desde la configuración actual del autómata.

La notación $LL(1)$ significa lo siguiente:

- **L**: la entrada se examina de izquierda a derecha (left to right).
- **L**: se construye la derivación más a la izquierda (leftmost derivation).
- **1**: sólo se necesita examinar un token de anticipación para decidir qué regla aplicar.

El concepto de gramática $LL(1)$ se puede extender a las gramáticas $LL(k)$, en las que es necesario emplear k símbolos de anticipación para poder realizar un análisis descendente de forma determinista.

La clase de gramáticas $LL(1)$ es suficientemente expresiva para cubrir las construcciones sintácticas de la mayoría de los lenguajes de programación. No obstante, dado un lenguaje cualquiera, no es posible determinar de forma automática si existe una gramática de tipo $LL(1)$ que lo genere, es decir, el problema de saber si un lenguaje es o no es $LL(1)$ es un *problema indecidible*.

Dada una gramática libre de contexto genérica, *a veces* es posible emplear algunas de las transformaciones expuestas en las secciones anteriores para obtener una gramática equivalente que sea $LL(1)$. En concreto, pueden resultar útiles las siguientes transformaciones:

- Eliminar la ambigüedad.
- Eliminar la recursividad por la izquierda.
- Factorizar por la izquierda.

Es interesante hacer una comparación entre la potencia de las gramáticas $LR(k)$ y las $LL(k)$. Para que una gramática sea $LR(k)$ debe ser posible reconocer la parte derecha de una producción en una forma sentencial derecha con la ayuda de k símbolos que aparecen a continuación. Este requisito es mucho menos restrictivo que el de las gramáticas $LL(k)$, con las que es necesario determinar la regla de producción con la que hay que derivar una cadena que comienza con k símbolos. Por ello, las gramáticas $LR(k)$ pueden describir más lenguajes que las $LL(k)$, formando los lenguajes generados por las segundas un subconjunto propio de los generados por las primeras.

3.7.2. Análisis descendente predictivo no recursivo

En esta sección se estudia la construcción de analizadores descendentes predictivos denominados *no recursivos* por el hecho de no apoyarse en el uso de funciones recursivas para gestionar la memoria de los pasos del análisis. Por el contrario, emplean con este fin una pila de forma explícita.

Estos analizadores hacen uso de una tabla de análisis que indica de forma determinista cuál es la regla de producción que se debe aplicar en la configuración actual. La configuración actual establece cuál es el nodo del árbol de análisis en el que se encuentra el analizador sintáctico (almacenado en la cima de la pila), y qué terminal se está leyendo en la entrada. Con esta información y la tabla de análisis, el analizador sintáctico puede determinar la regla de producción que se debe usar para realizar el siguiente paso de derivación y expandir así el árbol de análisis. En el caso de que el analizador se encuentre en un nodo hoja, se debe verificar que el terminal que etiqueta dicho nodo coincide con el terminal que se está leyendo en la entrada.

3.7.2.1. Construcción de la tabla de análisis

Los conjuntos PRIMERO y SIGUIENTE son necesarios para formalizar el concepto de gramática $LL(1)$, puesto que guían el funcionamiento del análisis descendente predictivo. Éstos, de utilidad también en los métodos ascendentes, se estudiaron en la sección 3.4.

Para la construcción de la tabla de análisis, es interesante la definición de una función denominada *Predict*, que puede definirse en función de PRIMERO y SIGUIENTE, mediante la cual se obtiene el conjunto de terminales cuya aparición en la entrada predicen el uso de una regla de producción de la gramática:

Definición 3.33. *La función* ***Predict*** *calcula, dada una regla de producción, los símbolos terminales que predicen su uso:*

$$\begin{aligned} \textit{Predict}(A \to \alpha) = \; & \textit{if } \lambda \in \textit{PRIMERO}(\alpha) \\ & \textit{then } (\textit{PRIMERO}(\alpha) - \{\lambda\} \cup \textit{SIGUIENTE}(A)) \\ & \textit{else PRIMERO}(\alpha) \end{aligned}$$

Obsérvese que la función *Predict* define, por cada regla de producción, un conjunto formado por símbolos terminales, incluido el símbolo fin de cadena \$. Nunca se incluye en este conjunto λ.

Dada una regla de producción $A \to \alpha$, los terminales que pueden aparecer al comienzo de las cadenas derivables desde α están incluidos en el conjunto *Predict* de esta regla. Estos terminales son, precisamente, el conjunto PRIMERO(α). Hay un caso especial, que corresponde a la situación en la que α puede derivar en λ. Cuando sucede esto, la aparición en la entrada de un terminal perteneciente a SIGUIENTE(A) también está prediciendo el uso de la regla de producción.

Definición 3.34. *La **tabla de análisis** de un analizador descendente predictivo no recursivo se define como:*

$$M : V_N \times (V_T \cup \{\$\}) \to 2^P$$

es decir, dado un no terminal que debe derivarse en una derivación más a la izquierda, y dado un símbolo terminal o el final de la entrada, la tabla indica un conjunto de reglas que pueden aplicarse para realizar la siguiente derivación.

Para construir la tabla de análisis se emplea la función *Predict* de la siguiente forma:

$$M[A, a] = \{A \to \alpha \mid a \in \text{Predict}(A \to \alpha)\}\ \forall A \in V_N, a \in (V_T \cup \{\$\})$$

La tabla de análisis de una gramática libre de contexto puede presentar varias reglas de producción en una entrada. Obviamente, si estamos interesados en un analizador predictivo, esta situación no es deseable. No obstante, una gramática $LL(1)$ no dará lugar a esta indeterminación.

Definición 3.35. *Una gramática libre de contexto $G = (V_N, V_T, S, P)$ es de tipo $\mathbf{LL(1)}$ si y sólo si, dadas cualquier par de producciones de un no terminal $A \to \alpha$ y $A \to \beta$, se cumple la condición:*

$$\textit{Predict}(A \to \alpha) \cap \textit{Predict}(A \to \beta) = \emptyset$$

Obsérvese que la condición expresada en la definición anterior es equivalente a que no existan entradas en la tabla de análisis con más de una regla de producción.

Algoritmo 3.19. Cálculo de la tabla de análisis sintáctico descendente predictivo.
Entrada: una gramática libre de contexto $G = (V_N, V_T, S, P)$, y sus conjuntos PRIMERO y SIGUIENTE.
Salida: Tabla de análisis M.

---

1. Para cada regla de producción $A \to \alpha$ de la gramática:
   1.1. Calcular el conjunto Predict($A \to \alpha$).
   1.2. Para cada terminal $a \in$ Predict($A \to \alpha$), añádase $A \to \alpha$ a $M[A, a]$.
2. Hágase que cada entrada no definida en M sea *error*.

Ejemplo 3.27. Se aplica el algoritmo de construcción de la tabla de análisis descendente predictivo a la gramática obtenida al final del ejemplo 3.26:

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

Obtenemos a continuación los conjuntos PRIMERO de todos los símbolos, empleando el algoritmo 3.1:

- Los conjuntos PRIMERO para todos los símbolos terminales de $V_T = \{(,), +, *\}$ son ellos mismos.
- Para el no terminal F, aplicando el paso 2.2 introducimos en su conjunto PRIMERO los símbolos (e id.
- Para el no terminal T', aplicando el paso 2.1 introducimos en su conjunto PRIMERO λ, y por el paso 2.2 el símbolo $*$.
- Para el no terminal T, por el paso 2.2, añadimos a su conjunto PRIMERO todos los símbolos de PRIMERO(F) por la regla $T \to FT'$. Como $\lambda \notin$ PRIMERO(F), no se añaden más símbolos.
- Para el no terminal E', con el paso 2.1 se añade λ a su conjunto PRIMERO, y con el paso 2.2 se añade $+$.
- Para el no terminal E, por el paso 2.2 añadimos a su conjunto PRIMERO todo el contenido de PRIMERO(T), por la regla $E \to TE'$. Como $\lambda \notin$ PRIMERO(T), no se añaden más símbolos.

Por el razonamiento anterior, se obtienen los conjuntos PRIMERO:

$$
\begin{aligned}
\text{PRIMERO}(F) &= \{(, id\} \\
\text{PRIMERO}(T') &= \{*, \lambda\} \\
\text{PRIMERO}(T) &= \{(, id\} \\
\text{PRIMERO}(E') &= \{+, \lambda\} \\
\text{PRIMERO}(E) &= \{(, id\}
\end{aligned}
$$

Pasamos ahora a calcular los conjuntos SIGUIENTE:

- Para el no terminal E, el conjunto SIGUIENTE incluye el símbolo \$ por el paso 1, y el símbolo) por el paso 2 y la producción $F \to (E)$.
- Para el no terminal E', añadimos todos los símbolos de SIGUIENTE(E) por la aplicación del paso 3 y la producción $E \to TE'$. Obsérvese que no tiene sentido el uso de la producción $E' \to +TE'$ para calcular SIGUIENTE(E'), puesto que conduce a una definición circular.
- Para el no terminal T, se añade el contenido de PRIMERO(E') – salvo el símbolo λ – a su conjunto SIGUIENTE por el paso 2 y la producción $E \to TE'$. Obsérvese que la producción $E' \to +TE'$ conduce a la misma acción. Además, como $\lambda \in$ PRIMERO(E'), añadimos el contenido de SIGUIENTE(E) y SIGUIENTE(E') por estas dos reglas de producción, usando el paso 3.
- Para el no terminal T', usando la regla $T \to FT'$ y aplicando el paso 3, añadimos a su conjunto SIGUIENTE los símbolos pertenecientes a SIGUIENTE(T).
- Para el no terminal F, usando las producciones $T \to FT'$ y $T' \to *FT'$, añadimos PRIMERO(T') – menos λ – a su conjunto SIGUIENTE, aplicando el paso 2. Por cumplirse $\lambda \in$ PRIMERO(T'), también añadimos a su conjunto SIGUIENTE los símbolos de SIGUIENTE(T) y SIGUIENTE(T').

Los conjuntos SIGUIENTE obtenidos son los siguientes:

$$
\begin{aligned}
\text{SIGUIENTE}(E) &= \{\$,)\} \\
\text{SIGUIENTE}(E') &= \{\$,)\} \\
\text{SIGUIENTE}(T) &= \{+, \$,)\} \\
\text{SIGUIENTE}(T') &= \{+, \$,)\} \\
\text{SIGUIENTE}(F) &= \{*, +, \$,)\}
\end{aligned}
$$

Con los conjuntos PRIMERO y SIGUIENTE se puede calcular la función Predict, obteniendo la siguiente tabla:

NO TERMINAL	SÍMBOLO DE ENTRADA					
	id	$+$	$*$	$($	$)$	\$
E	$E \to TE'$			$E \to TE'$		
E'		$E' \to +TE'$			$E' \to \lambda$	$E' \to \lambda$
T	$T \to FT'$			$T \to FT'$		
T'		$T' \to \lambda$	$T' \to *FT'$		$T' \to \lambda$	$T' \to \lambda$
F	$F \to id$			$F \to (E)$		

Se describe a continuación la aplicación del algoritmo para algunas reglas de producción:

- Puesto que Predict$(E \to TE') = \{(, id\}$, esta regla se introduce en las entradas $M[E, (]$ y $M[E, id]$.
- En el caso de $E' \to +TE'$, Predict$(E' \to +TE) = \{+\}$, de modo que la regla sólo se introduce en $M[E', +]$.
- Dado que $\lambda \in$ PRIMERO(λ), Predict$(E' \to \lambda) =$ SIGUIENTE$(E') = \{), \$\}$. La regla, por tanto, se introduce en las entradas $M[E',)]$ y $M[E', \$]$.

3.7.2.2. Algoritmo de análisis

En esta sección se describe el diseño del algoritmo que emplea el analizador descendente predictivo no recursivo para reconocer la cadena de entrada. Como se ha indicado previamente, la herramienta fundamental del análisis sintáctico es un autómata de pila. El tipo de analizador descendente no recursivo que estudiamos en esta sección hace un uso explícito de la pila. En la sección anterior se estudió un tipo de analizador descendente que emplea llamadas a funciones recursivas, usando implícitamente la pila.

La figura 3.6 muestra el diseño básico del analizador no recursivo.

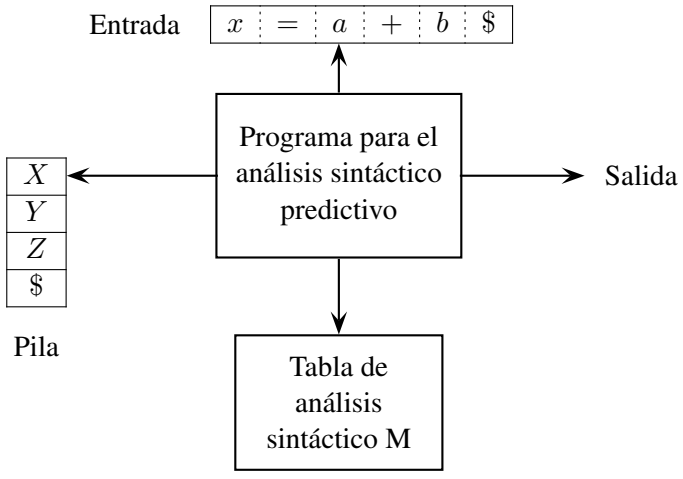


Figura 3.6: Modelo de un analizador predictivo no recursivo.

El analizador calcula una derivación más a la izquierda de la cadena de entrada. Si w es la porción de la entrada reconocida hasta ahora, el contenido de la pila es una secuencia α de símbolos de la gramática tal que:

$$S \Rightarrow^*_{mi} w\alpha$$

Es decir, la pila guarda la porción de la forma sentencial que está pendiente de ser derivada. En la cima de la pila se encuentra el símbolo más a la izquierda de α.

El analizador está controlado por un programa que, en cada iteración, considera el símbolo de la cima de la pila X y el siguiente símbolo a de la entrada. En función de su valor, realiza alguna de las operaciones siguientes:

- Si tanto X como a son el símbolo \$, el análisis finaliza con éxito, ya que la pila se ha vaciado y se ha reconocido toda la entrada.
- Si X es un terminal y coincide con a, el analizador saca X de la pila y desplaza el apuntador de la entrada un lugar a la derecha. Esta operación representa un reconocimiento correcto del terminal actual de la entrada.
- Si X es un terminal y no coincide con a, se señaliza que se ha producido un error, ya que la cadena no forma parte del lenguaje.
- Si X es un no terminal, el analizador debe consultar la entrada de la tabla $M[X, a]$:
  - Si $M[X, a]$ contiene la producción $X \to Y_1 Y_2 \dots Y_k$, el analizador extrae de la pila el símbolo X, e introduce en su lugar los símbolos del lado derecho de la producción, en orden inverso: Y_k,$Y_{k-1}, \dots, Y_1$. La salida es una indicación de que se ha usado dicha regla de producción. Este paso representa la expansión del árbol de análisis con los nodos Y_i por debajo del nodo X.
  - Si $M[X, a]$ está vacía, el análisis es incorrecto y la cadena de entrada no forma parte del lenguaje. La salida es una indicación de error.

A continuación se formaliza el algoritmo. En el pseudocódigo se emplea un apuntador al siguiente carácter de la entrada, denominado ae, una variable a para almacenar el carácter apuntado por ae, y una variable X para almacenar el símbolo del tope de la pila. También se hace uso de dos métodos de tratamiento de errores que se estudian más adelante.

Algoritmo 3.20. Algoritmo de análisis sintáctico descendente predictivo.
Entrada: una cadena w y la tabla de análisis M de una gramática $G = (V_N, V_T, S, P)$.
Salida: Si $w \in L(G)$, devuelve una derivación más a la izquierda de w. Si no, mensajes de error.

---

1. Inicialmente el analizador está en una configuración en la que tiene \$$S$ en la pila, con S en la cima, y w\$ en la entrada.
2. Hacer que ae apunte al primer símbolo de w\$ y asignarselo a a;
3. Asignar a X el símbolo del tope de la pila;
4. REPETIR:
   4.1. Si $(X == a)$ sacar X del tope de la pila y avanzar ae;
   4.2. en otro caso, si (X es terminal) llamar a `error1()`;
   4.3. en otro caso, si ($M[X, a]$ está vacía) llamar a `error2(X)`;
   4.4. en otro caso, si ($M[X, a] = X \to Y_1 Y_2 \dots Y_k$)
      4.4.1. Extraer X del tope de la pila;
      4.4.2. Meter Y_k,Y_{k-1},...,Y_1 en la pila, con Y_1 en la cima;
      4.4.3. Emitir la producción $X \to Y_1 Y_2 \dots Y_k$;
   4.5. Asignar a X el símbolo del tope de la pila y a a el símbolo apuntado por ae;

   HASTA $(X ==$ \$);

Ejemplo 3.28. Para mostrar la aplicación del algoritmo, empleamos la gramática siguiente:

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

cuya tabla de análisis $LL(1)$ se obtuvo en el ejercicio 3.27. Por comodidad, se repite a continuación:

NO TERMINAL	SÍMBOLO DE ENTRADA					
	id	$+$	$*$	$($	$)$	\$
E	$E \to TE'$			$E \to TE'$		
E'		$E' \to +TE'$			$E' \to \lambda$	$E' \to \lambda$
T	$T \to FT'$			$T \to FT'$		
T'		$T' \to \lambda$	$T' \to *FT'$		$T' \to \lambda$	$T' \to \lambda$
F	$F \to id$			$F \to (E)$		

Seguidamente se muestra la traza de aplicación del algoritmo para el reconocimiento de la cadena $id + id * id$:

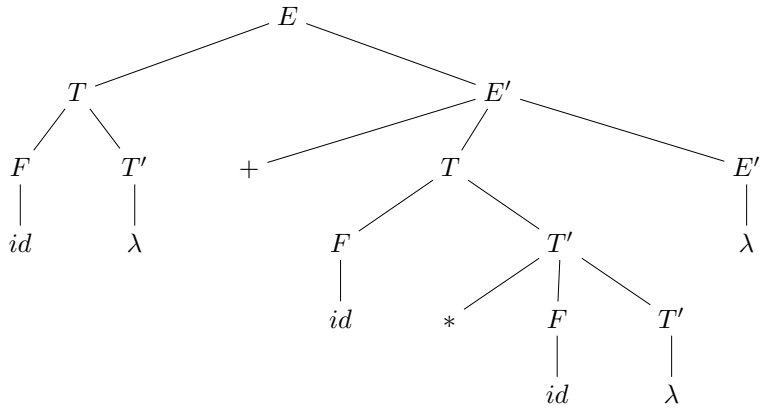
PILA	ENTRADA	SALIDA
\$ E	$id + id * id$ \$	
\$ E' T	$id + id * id$ \$	$E \to TE'$
\$ E' T' F	$id + id * id$ \$	$T \to FT'$
\$ E' T' id	$id + id * id$ \$	$F \to id$
\$ E' T'	$+ id * id$ \$	
\$ E'	$+ id * id$ \$	$T' \to \lambda$
\$ E' T $+$	$+ id * id$ \$	$E' \to +TE'$
\$ E' T	$id * id$ \$	
\$ E' T' F	$id * id$ \$	$T \to FT'$
\$ E' T' id	$id * id$ \$	$F \to id$
\$ E' T'	$* id$ \$	
\$ E' T' F $*$	$* id$ \$	$T' \to *FT'$
\$ E' T' F	id \$	
\$ E' T' id	id \$	$F \to id$
\$ E' T'	\$	
\$ E'	\$	$T' \to \lambda$
\$	\$	$E' \to \lambda$

La primera configuración es la indicada en el punto 1 del algoritmo. La pila contiene $\$E$, ya que E es el símbolo inicial de la gramática. La entrada es la cadena que hay que reconocer, seguida del símbolo \$. En la primera iteración del algoritmo $X = E$ y $a = id$. Se aplica el punto 4.4 del algoritmo, encontrando la producción $E \to TE'$ en la entrada $M[E, id]$. Se extrae E de la pila y se introduce en su lugar la parte derecha de la producción, en orden inverso, quedando la pila con $\$E'T$. Finalmente se emite la producción $E \to TE'$. El nuevo estado de la pila, la entrada, y la producción emitida, quedan reflejados en una nueva configuración del analizador.

La derivación más a la izquierda obtenida al finalizar el análisis es:

$$
\begin{aligned}
&E \Rightarrow TE' \Rightarrow FT'E' \Rightarrow idT'E' \Rightarrow idE' \Rightarrow id + TE' \Rightarrow id + FT'E' \Rightarrow id + idT'E' \\
&\Rightarrow id + id * FT'E' \Rightarrow id + id * idT'E' \Rightarrow id + id * idE' \Rightarrow id + id * id
\end{aligned}
$$

y el árbol de derivación es el siguiente:



Es interesante comprobar que la derivación más a la izquierda corresponde a la construcción del árbol de análisis realizando un recorrido en preorden.

3.7.2.3. Ambigüedad en los analizadores descendentes predictivos no recursivos

En la sección 3.2.2 se estudió el problema de la ambigüedad de algunas gramáticas que expresan parte de la sintaxis de los lenguajes de programación. En esta sección se plantea este problema desde el punto de vista del diseño de un analizador descendente predictivo no recursivo.

La gramática no ambigua de las sentencias `if-then` e `if-then-else`, obtenida en 3.2.2.2, es:

$$
\begin{aligned}
sent &\rightarrow sent_e \\
&\mid sent_{ne} \\
sent_e &\rightarrow \textbf{if}\ expr\ \textbf{then}\ sent_e\ \textbf{else}\ sent_e \\
&\mid \textbf{s} \\
sent_{ne} &\rightarrow \textbf{if}\ expr\ \textbf{then}\ sent \\
&\mid \textbf{if}\ expr\ \textbf{then}\ sent_e\ \textbf{else}\ sent_{ne} \\
expr &\rightarrow \textbf{e}
\end{aligned}
$$

Comprobamos, a continuación, si esta gramática puede emplearse para construir un analizador $LL(1)$. En primer lugar, se debe eliminar el factor común **if** $expr$ **then** en las reglas de $sent_{ne}$:

$$
\begin{aligned}
sent &\rightarrow sent_e \\
&\mid sent_{ne} \\
sent_e &\rightarrow \textbf{if}\ expr\ \textbf{then}\ sent_e\ \textbf{else}\ sent_e \\
&\mid \textbf{s} \\
sent_{ne} &\rightarrow \textbf{if}\ expr\ \textbf{then}\ sent'_{ne} \\
sent'_{ne} &\rightarrow sent \\
&\mid sent_e\ \textbf{else}\ sent_{ne} \\
expr &\rightarrow \textbf{e}
\end{aligned}
$$

En esta nueva gramática podemos observar que:

$$
\begin{aligned}
\text{PRIMERO}(sent_e) &= \{\textbf{if}, \textbf{s}\} \\
\text{PRIMERO}(sent_{ne}) &= \{\textbf{if}\} \\
\text{Predict}(sent \rightarrow sent_e) &= \{\textbf{if}, \textbf{s}\} \\
\text{Predict}(sent \rightarrow sent_{ne}) &= \{\textbf{if}\}
\end{aligned}
$$

de modo que la intersección de los conjuntos Predict de ambas reglas no es vacía. Eso implica que la gramática, aún no siendo ambigua, no es $LL(1)$.

En casos como este, podemos emplear una convención que consiste en:

- Partir de la gramática inicial, ambigua pero más sencilla.
- Factorizar la gramática en el caso de que sea posible.
- Construir la tabla de análisis aún cuando vaya a presentar conflictos (entradas con varias reglas).
- Seleccionar la regla que resuelve cada conflicto de acuerdo con un criterio que se desea forzar.

La gramática inicial de las sentencias `if-then` e `if-then-else` es:

$$\begin{array}{rcl} sent & \rightarrow & \textbf{if } expr \textbf{ then } sent \\ & | & \textbf{if } expr \textbf{ then } sent \textbf{ else } sent \\ & | & \textbf{s} \\ expr & \rightarrow & \textbf{e} \end{array}$$

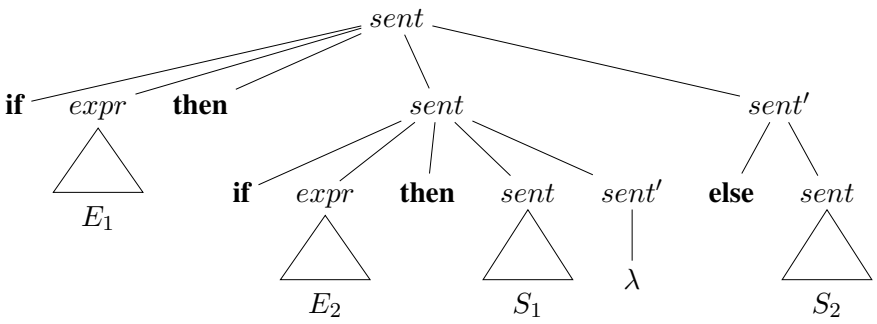
Factorizando esta gramática obtenemos:

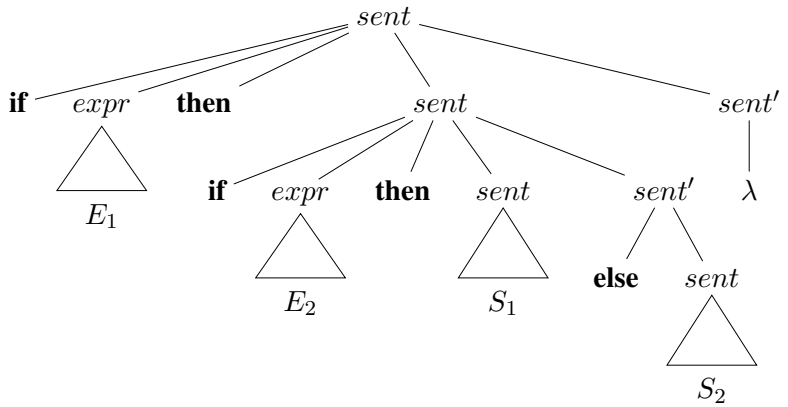
$$\begin{array}{rcl} sent & \rightarrow & \textbf{if } expr \textbf{ then } sent\ sent' \\ & | & \textbf{s} \\ sent' & \rightarrow & \textbf{else } sent \\ & | & \lambda \\ expr & \rightarrow & \textbf{e} \end{array}$$

Esta gramática sigue siendo ambigua, como se puede comprobar construyendo los árboles de derivación de una sentencia como:

$$\textbf{if } E_1 \textbf{ then if } E_2 \textbf{ then } S_1 \textbf{ else } S_2$$

tiene dos árboles de derivación:





Como ya sabemos, el convenio que se suele adoptar es asociar el **else** al **if** más próximo, siendo el segundo árbol el que cumple este criterio.

La tabla de análisis de la gramática es la siguiente:

NO TERMINAL	SÍMBOLO DE ENTRADA					
	if	**then**	**else**	**s**	**e**	$
$sent$	$sent \rightarrow$ **if** $expr$ **then** $sent\ sent'$			$sent \rightarrow$ **s**		
$sent'$			$sent' \rightarrow$ **else** $sent$ $sent' \rightarrow \lambda$			$sent' \rightarrow \lambda$
$expr$					$expr \rightarrow$ **e**	

El conflicto se produce porque:

$$\text{Predict}(sent' \rightarrow \textbf{else}\ sent) = \{\textbf{else}\}$$
$$\text{Predict}(sent' \rightarrow \lambda) = \{\textbf{else}, \$\}$$

Por convenio, al consultar la entrada $M[sent', \textbf{else}]$, se utilizará la producción $sent' \rightarrow$ **else** $sent$, que corresponde a asociar el **if** con el **else** más próximo.

No obstante, en general no hay ninguna forma sistemática para elegir una única regla de producción en los conflictos debidos a la ambigüedad de la gramática. Por ello puede ser necesario alterar el lenguaje para eliminar la ambigüedad.

3.7.3. Análisis descendente predictivo recursivo

Este método descendente de análisis es alternativo al método no recursivo (sección 3.7.2). Se basa en la ejecución, de forma recursiva, de un conjunto de procedimientos que se encargan de procesar la entrada. Cada no terminal de la gramática tiene asociado un procedimiento, cuyo código depende del lado derecho de las reglas de producción de dicho no terminal. Por tanto, un analizador predictivo recursivo se construye de forma totalmente dependiende de la gramática.

Al emplear gramáticas $LL(1)$, el método de cada no terminal puede apoyarse en el uso de la función Predict para decidir cuál es el lado derecho que se debe emplear en cada momento. De forma genérica, si A es un no terminal de la gramática, el pseudocódigo del método que se le asocia es el siguiente:

```
1 void coincide(terminal t) {
2         if (preanalisis == t) preanalisis = sigTerminal();
3         else error1();
4 }
5 void A() {
```

```
6       // Sea A → α1 | α2 | ... | αn
7       if (preanalisis ∈ Predict(αi)) para algún i tal que αi ≡ X1X2...Xk
8               for (j = 1 to k) {
9                       if (Xj ∈ VN) Xj();
10                      else coincide(Xj);
11              }
12      else error2(A);
13  }
```

donde `preanalisis` es una variable que contiene el token actual. El método `coincide` se encarga de verificar que el token actual coincide con el terminal que se pasa como argumento, y en caso de ser así, se avanza al siguiente token. El código anterior incluye llamadas a dos métodos de error, `error1()` y `error2(X)`, que ya vimos en el algoritmo del analizador no recursivo. Su función es la misma para ambos analizadores.

Obsérvese que el mecanismo de llamadas recursivas está gestionando implícitamente una pila mediante la cual se memoriza el estado del proceso de análisis.

Ejemplo 3.29. Construimos una analizador descendente predictivo recursivo para la siguiente gramática libre de contexto:

$$
\begin{array}{rcl}
\text{TIPO} & \rightarrow & \text{SIMPLE} \\
 & | & \uparrow \text{id} \\
 & | & \text{array [SIMPLE] of TIPO} \\
\text{SIMPLE} & \rightarrow & \text{integer} \\
 & | & \text{char} \\
 & | & \text{num puntopunto num}
\end{array}
$$

Se trata de una gramática que define tipos de datos compuestos y tipos simples. Los compuestos son punteros a identificadores y arrays de tipos compuestos. Los simples son enteros, caracteres y secuencias de números. En la definición de la gramática, todos los términos en minúsculas pertenecen a V_T. Es fácil observar que esta gramática es $LL(1)$ ya que los respectivos conjuntos Predict son disjuntos. Por lo tanto, el símbolo de entrada va a determinar qué producción aplicar de forma determinista.

Los métodos de los dos no terminales de la gramática son:

```
1   void TIPO() {
2       if (preanalisis∈{'integer','char','num'}) SIMPLE();
3       else if (preanalisis=='↑') {
4               coincide('↑'); coincide('id');
5       }
6       else if (preanalisis=='array') {
7               coincide('array'); coincide('['); SIMPLE(); coincide(']');
8               coincide('of'); TIPO();
9       }
10      else error2('TIPO');
11  }
12  void SIMPLE() {
13      if (preanalisis=='integer') coincide('integer');
14      else if (preanalisis=='char') coincide('char');
15      else if (preanalisis=='num') {
16              coincide('num'); coincide('puntopunto'); coincide('num');
17      }
18      else error2('SIMPLE');
19  }
```

3.7.4. Tratamiento de errores en el análisis descendente predictivo

Los analizadores $LL(1)$ son capaces de detectar un error sintáctico en cuanto se produce, y pueden repararlo sin hacer retroceso en la entrada. Tanto en los analizadores recursivos como en los no recursivos se pueden producir dos tipos de errores similares, identificados en el pseudocódigo de los algoritmos mostrados anteriormente como `error1` y `error2`. En los siguientes apartados se describen estrategias para solucionarlos de forma adecuada a cada tipo de analizador.

3.7.4.1. Tratamiento de errores en analizadores no recursivos

Los dos tipos de errores se pueden identificar de la siguiente forma:

- El terminal del tope de la pila no coincide con el token de la entrada (`error1`).
- En la pila hay un no terminal a expandir, y en la tabla no aparece ninguna producción aplicable para esa variable y el símbolo de entrada (`error2`).

Existen dos estrategias para tratar los errores sintácticos: la recuperación a nivel de frase y la recuperación en modo pánico.

Recuperación a nivel de frase.

El esquema general de tratamiento de errores con esta técnica consiste en introducir apuntadores a rutinas de error en las casillas en blanco de la tabla M. Dependiendo de cual sea la casilla de error, la rutina de tratamiento ejecutará un tipo de operación u otro. Las operaciones habituales son las de cambiar, eliminar o añadir caracteres a la entrada emitiendo los pertinentes mensajes de error. Este enfoque puede resultar bastante complicado, pues habría que considerar los posibles símbolos de entrada que pueden causar error, realizar el tratamiento adecuado para cada tipo de error y emitir un mensaje de error pertinente.

Recuperación en modo pánico.

La recuperación de errores en modo pánico se activa cuando se detecta un token no esperado, y procede consumiendo tokens hasta que se recibe un determinado token perteneciente a un conjunto de tokens de *sincronización*. Los tokens de sincronización forman un conjunto que debe ser seleccionado cuidadosamente pues la eficiencia del manejo de errores en modo pánico depende, en gran medida, de su elección.

La estrategia básica de este tipo de recuperación de errores consiste en:

- Si $M[A, a] = \emptyset$, se descartan tokens de la entrada hasta recibir un nuevo token a tal que se cumpla alguna de las siguientes condiciones:
  - $a \in$ PRIMERO(A). En este caso, se puede continuar el análisis derivando el no terminal A, puesto que esta condición implica que está definida la entrada $M[A, a]$.
  - $a \in$ SIGUIENTE(A). Esta situación permite sacar A de la pila, simulando que ha sido reconocida una cadena de tokens derivable de A. Se continúa el análisis con normalidad tras esta modificación.
- Si no coinciden el terminal de la entrada con el de la cima de la pila, extraemos el terminal de la cima de la pila y continuamos el análisis. Esta acción simula la inserción del terminal de la cima de la pila en la entrada.

Ejemplo 3.30. Partiendo de la tabla de análisis del ejercicio 3.28, se va a aplicar el método de recuperación de errores en modo pánico al análisis de la entrada $+id * id+$.

PILA	ENTRADA	ACCIÓN
$\$\ E$	$+\ id * id + \$$	**Error**: ignorar $+$ al no ser token de sincronización.
$\$\ E$	$id * id + \$$	Como $id \in$ PRIMERO(E), continuar el análisis de modo normal.
$\$\ E'\ T$	$id * id + \$$	$E \to TE'$
$\$\ E'\ T'\ F$	$id * id + \$$	$T \to FT'$
$\$\ E'\ T'\ id$	$id * id + \$$	$F \to id$
$\$\ E'\ T'$	$* id + \$$	
$\$\ E'\ T'\ F\ *$	$* id + \$$	$T' \to *FT'$
$\$\ E'\ T'\ F$	$id + \$$	
$\$\ E'\ T'\ id$	$id + \$$	$F \to id$
$\$\ E'\ T'$	$+ \$$	
$\$\ E'$	$+ \$$	$T' \to \lambda$
$\$\ E'\ T\ +$	$+ \$$	$E' \to +TE'$
$\$\ E'\ T$	$\$$	**Error**: se extrae T de la pila, ya que $\$ \in$ SIGUIENTE(T)
$\$\ E'$	$\$$	
$\$$	$\$$	$E' \to \lambda$

Se puede observar que hay una primera secuencia de derivaciones más a la izquierda, antes de la detección del segundo error:

$$E \Rightarrow TE' \Rightarrow FT'E' \Rightarrow idT'E' \Rightarrow id * FT'E' \Rightarrow id * idT'E' \Rightarrow id * idE' \Rightarrow id * id + TE'$$

A partir de este punto, no se puede seguir generando la cadena. Eliminando T de la cima de la pila se puede continuar aplicando la regla $E' \Rightarrow \lambda$, obteniendo la última derivación:

$$id * id + E' \Rightarrow id * id+$$

De modo que el algoritmo *simula* la derivación de la cadena errónea.

Veamos un segundo ejemplo con la entrada $(id\$$. La evolución del algoritmo es:

PILA	ENTRADA	ACCIÓN
$\$\ E$	$(\ id\ \$$	
$\$\ E'\ T$	$(\ id\ \$$	$E \to TE'$
$\$\ E'\ T'\ F$	$(\ id\ \$$	$T \to FT'$
$\$\ E'\ T'\ )\ E\ ($	$(\ id\ \$$	$F \to (E)$
$\$\ E'\ T'\ )\ E$	$id\ \$$	
$\$\ E'\ T'\ )\ E'\ T$	$id\ \$$	$E \to TE'$
$\$\ E'\ T'\ )\ E'\ T'\ F$	$id\ \$$	$T \to FT'$
$\$\ E'\ T'\ )\ E'\ T'\ id$	$id\ \$$	$F \to id$
$\$\ E'\ T'\ )\ E'\ T'$	$\$$	
$\$\ E'\ T'\ )\ E'$	$\$$	$T' \to \lambda$
$\$\ E'\ T'\ )$	$\$$	$E' \to \lambda$
$\$\ E'\ T'$	$\$$	**Error**: se extrae $)$ de la pila.
$\$\ E'$	$\$$	$T' \to \lambda$
$\$$	$\$$	$E' \to \lambda$
$\$$	$\$$	

El algoritmo en modo pánico ha interpretado que se había omitido el paréntesis derecho. Esa interpretación produce la derivación izquierda siguiente:

$$E \Rightarrow TE' \Rightarrow FT'E' \Rightarrow (E)T'E' \Rightarrow (TE')T'E' \Rightarrow (FT'E')T'E'$$
$$\Rightarrow (idT'E')T'E' \Rightarrow (idE')T'E' \Rightarrow (id)T'E' \Rightarrow (id)E' \Rightarrow (id)$$

3.7.4.2. Tratamiento de errores en analizadores recursivos

Se puede implementar la recuperación en modo pánico en los analizadores recursivos, de forma equivalente a la vista para los analizadores no recursivos, mediante la implementación de los métodos `error1()` y `error2(`$X \in V_N$`)` usados en el algoritmo descrito en la sección 3.7.3. La implementación debe hacerse del siguiente modo:

- `error1()`: el método debe informar del error, pero no tiene que realizar ninguna otra operación. Esto equivale a sacar el terminal de la pila, como si se hubiese reconocido en la entrada.
- `error2(`$X \in V_N$`)`: además de informar del error, el método debe consumir símbolos de la entrada hasta que se cumpla una de las siguientes condiciones:
  - El siguiente token de la entrada pertenece a PRIMERO(X). En este caso, se vuelve a llamar al método de no terminal `X()`.
  - El siguiente token de la entrada pertenece a SIGUIENTE(X). En este caso, se sale del procedimiento sin hacer nada más.

3.8. Métodos universales de análisis sintáctico

Aunque no son objeto de estudio de esta asignatura, es interesante indicar que existen métodos de análisis sintáctico generales que permiten trabajar con cualquier tipo de gramática libre de contexto. Estos métodos suelen requerir ciertas transformaciones en las gramáticas antes de ser aplicados, transformaciones que siempre son posibles. No obstante, se emplean poco porque no son eficientes. A continuación se indican brevemente sus características principales:

- Método *Cocke-Younger-Kasami* (CYK). Para aplicarlo, la gramática debe encontrarse en *forma normal de Chomsky*[10]. El método consta de dos algoritmos. El primero crea una tabla de análisis que funciona de forma ascendente. El segundo devuelve las derivaciones por la izquierda (análisis descendente) de la cadena a analizar. El método requiere, para una gramática arbitraria, $O(n^3)$ en tiempo y $O(n^2)$ en espacio.
- Método *Earley*. También consta de dos algoritmos. El primero calcula un conjunto de listas de análisis, y el segundo devuelve las derivaciones por la derecha (análisis ascendente) de la cadena de entrada. Para aplicar el segundo algoritmo se requiere que la gramática de entrada no tenga ciclos. El algoritmo comprueba si $w \in L(G)$ con una complejidad $O(n^3)$ en tiempo y $O(n^2)$ es espacio, para una gramática arbitraria. Si la gramática no es ambigua, se puede conseguir un tiempo $O(n^2)$.
- *Análisis general descendente con retroceso*. En este método se requiere que la gramática no sea recursiva por la izquierda. Comenzando con el símbolo inicial de la gramática, el algoritmo busca un conjunto de derivaciones por la izquierda para generar la cadena de entrada, teniendo en cuenta todas las alternativas posibles en caso de ser necesario (la gramática puede ser ambigua). Este algoritmo tiene orden de complejidad lineal en espacio y exponencial en tiempo, en función de la longitud de la cadena de entrada $|w|$.
- *Análisis general ascendente con retroceso*. Se requiere que la gramática sea λ-libre y libre de ciclos. El algoritmo busca, a partir de la cadena de entrada, un conjunto de reducciones por la izquierda hasta llegar al símbolo inicial, teniendo en cuenta, si es necesario, todas las posibilidades. Este algoritmo tiene orden de complejidad lineal en espacio y exponencial en tiempo, también en función de la longitud de la cadena de entrada $|w|$.

[10] Una gramática libre de contexto está en forma normal de Chomsky si es λ-libre y sus producciones son de la forma $A \to BC$ o $A \to a$ donde $A, B, C \in V_N$ y $a \in V_T$.