Compiladores
Facultad de Ciencias UNAM
Análisis Sintáctico: *Parser bottom-up* [*]

Lourdes Del Carmen González Huesca [**]

24 de marzo de 2024

Resumen

Los compiladores se encargan de determinar si un programa está construido bajo las reglas establecidas en la especificación del lenguaje o gramática que lo define. El proceso de análisis sintáctico se encarga de verificar si un programa o cadena dado tiene un árbol de derivación bajo la gramática dada. En esta nota se exponen algunos métodos para determinar si dicho árbol de derivación existe o no para el código fuente en cuestión, es decir se determina si el código fuente pertenece al lenguaje generado por la gramática o no.

En particular se exploran los métodos utilizados por los parsers bottom-up, los cuales se caracterizan por construir el árbol de derivación empezando por las hojas y terminando en la raíz.

Términos clave:

Es: Parsers bottom-up, Handle/Asa/Mango, Parser LR, Parser LALR, Items, Tabla de Parsing LALR, Conjuntos canónicos, Parsing Shift-Reduce.

En: Bottom-up Parsing, Handle pruning, LR Parsing, LALR Parsing, LR(1) Sets of items, LALR Parsing Tables, Canonical LR Items, Shift-Reduce parsing.

Los analizadores *bottom-up* construyen un árbol de sintaxis concreta (*parse-tree*) desde las hojas y hacia la raíz, buscando reconocer partes derechas de las producciones para sustituirlas por símbolos no-terminales hasta obtener el símbolo inicial de la gramática. Esta técnica para generar los árboles se conoce como *shift-reduce* donde una reducción de símbolos se realiza para abstraer el proceso inverso de una derivación paso a paso. La clase de gramáticas que caracterizan a estos analizadores son las **LR** (lectura left-to-right de la cadena de entrada y haciendo derivaciones más a la derecha). Estas gramáticas son más expresivas que las **LL** ya que permiten recursión izquierda en las producciones. Veremos varios tipos de estas gramáticas y la construcción de las tablas de parsing para ejecutar el parser.

[*] Los ejemplos están tomados del libro "Compilers, Principles, Techniques and Tools" [1]

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

Preliminares

Las gramáticas que consideramos no son ambiguas para asegurar que existe una única derivación de la cadena de entrada. En este caso se realiza reescribiendo el símbolo no-terminal más a la derecha en el proceso de derivación.

El análisis **LR** funciona en general procesando la cadena y tomando la regla más adecuada de la gramática para construir un subárbol. Este analizador es un autómata de pila, donde la pila representa la historia de visita de estados, el tope de la pila es justo el estado actual. Este análisis se ayuda de una tabla de parsing para seleccionar la regla correspondiente mediante:

1. abstraer la gramática como un autómata finito no-determinista donde el estado indica una producción parcialmente reconocida;

2. transformar este autómata, de ser posible, en uno determinista.

Recordemos que la cadena de entrada es una secuencia de lexemas. Así, este tipo de parsing utiliza un autómata finito para construir la tabla de parsing y el propio parser, un autómata de pila, puede realizar una de dos acciones en un estado q considerando que la pila contiene a los estados precedentes:

- desplazamiento (*shift*)
  si el lexema inicial es a, eliminar a de la cadena entrada y guardar en la pila el nuevo estado q' obtenido por la función de transición junto con el lexema o símbolo leido; es decir que si se lee un símbolo terminal se guarda en la pila.

- reducir (*reduce*)
  si el estado q está etiquetado por $A \to \beta\bullet$, sacar de la pila el mismo número de símbolos que la longitud de β para regresar el autómata a un estado anterior p y después guardar el nuevo estado p' que se obtenga desde p al leer A; es decir que se reconoce el tope de la pila con la parte derecha de alguna producción y se reemplaza uno por el otro.

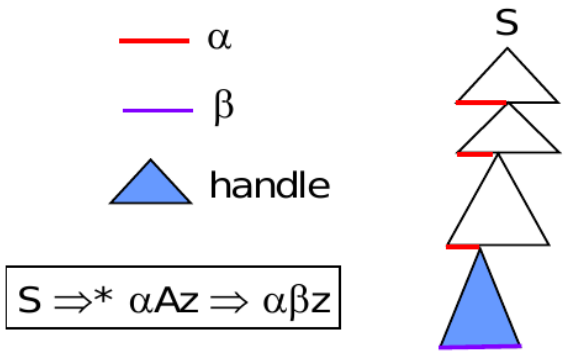


Figura 1: Imagen tomada de Notas del curso (CIS 341) - Compilers, Universidad de Pennsylvania [7].

Dada una cadena $\alpha\beta z$ (con z cadena de símbolos terminales), se define el **handle**, asa [1] o mango de la cadena como una subcadena que tiene el mismo patrón que la parte derecha de una producción $A \to \beta$ y cuya reducción es el símbolo no-terminal en la izquierda de la producción. Esto es lo que representa un paso en el proceso reverso de una derivación más a la derecha de una cadena y serán justo los símbolos en el tope de la pila para ser reducidos.

Los algoritmos que estudiaremos utilizan las operaciones shift-reduce para encontrar handles y construir el árbol. Para ello utilizaremos extensiones de los lenguajes tanto de la cadena de entrada al incluir un símbolo especial de fin de cadena o archivo (#, `eof`), como de la gramática al incluir un nuevo símbolo inicial (S, S' o cualquier otro determinado para ello) que debe establecerse para definir el parser.

Las configuraciones del parser o autómata de pila cambian respecto a cada método pero la configuración inicial debe tener por un lado el símbolo de fondo de la pila (usualmente # o `eof`) y por otro la cadena de entrada seguida de #. La configuración final debe de tener por un lado el símbolo inicial de la gramática extendida y por otro lado se consume la cadena de entrada y sólo resta el símbolo de fin de cadena.

Sin importar el tipo de parser bottom-up que revisemos, se tienen las siguientes ventajas:

- método más usual para reconocer gramáticas libres de contexto de lenguajes de programación;
- método más eficiente sin backtracking que utiliza shift-reduce;
- detecta errores más rápido al construir el árbol desde las hojas.

Una gramática ambigua no puede pertenecer a la clase **LR** y decimos que una gramática está en una clase o es de algún tipo si hay un analizador de esa clase que reconozca el lenguaje de dicha gramática.

En este tipo de analizadores también es posible considerar el procesamiento de símbolos extras que ayudan a determinar más rápido la regla del lenguaje que genera un subárbol. Los analizadores **LR(k)** reconocen las cadenas que puedan estar a la derecha de una producción y después se decide cuál es la producción indicada, de esta forma se revisan varias posibilidades de las ramas en paralelo. Se busca reconocer β_1 o β_2 de las producciones $A \to \beta_1$ y $A \to \beta_2$ además de leer por adelantado k símbolos.

El proceso importante dentro de estos analizadores es decidir cuándo realizar una operación shift o un reduce, esto se resuelve manteniendo estados en un autómata finito para recordar la posición en el análisis, es decir usar una tabla de acciones que determine lo que debe hacer el parser.

Puede suceder que un analizador shift-reduce alcance una configuración en donde no se pueda decidir si realizar un shift o un reduce (conflicto shift/reduce), o en donde haya diferentes reducciones (conflicto reduce/reduce) y decimos que las gramáticas que presenten conflictos no pertenecen a alguna clase **LR(k)**.

[1] En alfarería, las asas son elementos de suspensión y aprehensión, es decir: las partes de la pieza, por lo general curvas, que facilitan su manejo; denominación extensiva a todo tipo de recipientes. (Wikipedia, `https://es.wikipedia.org/w/index.php?title=Asa_(alfarer%C3%ADa)&oldid=123732093`)

Algoritmo general para parser LR

El algoritmo se sirve de la cadena de entrada y una tabla de parsing M que contiene acciones (shift, reduce o accept) y las transiciones entre estados. La tabla tiene parejas de estado-símbolo que representarán el estado actual del parser y el símbolo del input:

> Si $M[q, c] = \mathsf{shift}(q')$ entonces se realiza un shift y se almacena q' es decir $\text{push}(q')$
>
> Si $M[q, c] = A \rightarrow \alpha$ entonces se reduce al sacar a los elementos de la pila correspondientes a esa regla, es decir: $\text{pop}_{|\alpha|}$; $q = \text{top}$; $\text{push}(M[q', A])$

De acuerdo a diferentes tipos de gramáticas LR es que se genera la tabla de parsing, pero el algoritmo general para analizar una cadena es el siguiente:

Input: Cadena de entrada w y la tabla LR con las funciones ACTION y GOTO.

Output: Si la cadena w está en el lenguaje de la gramática, entonces se devuelven las reducciones del parsing bottom-up; sino se devuelve un error.

El parser inicia con el estado inicial s_0 en la pila y $w\#$ como la cadena de entrada (input)

```
let a be the first symbol of w#;
while(1)  /* repeat forever */
{
     let s be the state on top of the stack;
     if ( ACTION[s; a] = shift t )
     {
         push t onto the stack;
         let a be the next input symbol;
     } else if ( ACTION[s; a] = reduce A→v )
            {
              pop v symbols off the stack;
              let state t now be on top of the stack;
              push GOTO[t; A] onto the stack;
              output the production A→v;
            } else if ( ACTION[s; a] = accept ) break;
                   /* parsing is done */
            else call error-recovery routine;
}
```

Se considera que v es una cadena de símbolos de la gramática.

El comportamiento del algoritmo es el mismo para diferentes tipos de parsing LR, es decir el manejo del autómata de pila que depende de la tabla de parsing descrito en el algoritmo anterior. Por lo tanto, nos concentraremos en las variantes para la construcción de la tabla. Esta tabla se construye con apoyo de un autómata finito que decidirá cada valor en las entradas ella. Para ello es necesario extender la gramática del lenguaje con un símbolo inicial nuevo.

Veamos ahora los diferentes tipos de analizadores **LR**.

LR(0)

Para este método, se decidirá una acción (shift o reduce) sin considerar un símbolo leído por adelantado. La tabla de parsing definirá los estados de la máquina y las transiciones. Los estados de la máquina finita serán conjuntos de producciones de la gramática donde se ha identificado una subcadena, estas producciones utilizan una bandera para realizar esta identificación:

Definición 1 (Item)**.** Un item es una producción de la gramática con un punto o bandera ($\bullet$) en alguna posición del cuerpo o parte derecha. Este punto marca la coincidencia en el proceso de análisis para identificar los símbolos analizados. Se clasifican en dos:

- **item tipo kernel**: es el item de la producción inicial $S \to \bullet E$ (donde E es el símbolo inicial de la gramática original) o cualquier item que **no** tenga un punto más a la izquierda
- **item tipo no-kernel**: es cualquier item con el punto más a la izquierda excepto el item inicial $S \to \bullet E$

Ejemplo 1 (Items)**.** Consideramos una regla $A \to a_1 a_2 a_3 \dots a_n$ donde A es claramente una variable de la gramática y cada símbolo a_i es un terminal o una variable del lenguaje. Se tienen los siguientes items:

- $A \to \bullet a_1 a_2 a_3 \dots a_n$ item que indica que no se ha reconocido nada del cuerpo de la regla
- $A \to a_1 a_2 a_3 \dots a_i \bullet a_{i+1} \dots a_n$ item que indica que se ha reconocido desde a_1 hasta a_i
- $A \to a_1 a_2 a_3 \dots a_n \bullet$ item que indica que se ha reconocido todo el cuerpo de la regla

Los conjuntos de items son **canónicos** si son la base para construir un autómata finito para abstraer las decisiones de shift-reduce. Los estados y las transiciones se calculan con las siguientes funciones auxiliares:

Definición 2 (Closure en LR(0))**.** Sea I un conjunto de items, la cerradura de I o Closure(I) son los items tales que

1. es elemento de I
2. Si $A \to \alpha \bullet B\beta$ está en Closure(I) y $B \to \gamma$ es una producción de la gramática, entonces agregar $B \to \bullet\gamma$ a Closure(I)

El algoritmo es el siguiente:

```
let J = I;
repeat{
   for each item A → α • Bβ in J
       for each grammar production B → γ
          if (B → •γ not in J)
          then add B → •γ
}
until no more items are added to J
```

Definición 3 (Goto en LR(0))**.** Dado un conjunto de items I y un símbolo cualquiera de la gramática X, la función Goto del estado I mediante X es la cerradura de todos los items $A \to \alpha X \bullet \beta$ tal que $A \to \alpha \bullet X\beta$ está en I.

El algoritmo que construye el autómata finito con los conjuntos canónicos de items `C` y sus transiciones para una gramática extendida con el símbolo S es el siguiente:

```
C = CLOSURE({S → • E});
repeat{
   for each set of items I in C
       for each grammar symbol X
          if (GOTO(I,X) is not empty and not in C)
          then add GOTO(I,X) to C
}
until no new sets are added to C
```

Para llenar la tabla M se toman las transiciones del autómata $J = \text{GoTo}(I, X)$ con X cualquier símbolo de la gramática:

- Si X es terminal entonces $M(I, X) = sJ$ para denotar un shift de X y transitar al estado J.
- Si X es no-terminal entonces $M(I, X) = J$ para denotar una transición al estado J.
- Si I contiene $S' \to S\bullet$ entonces $M(I, \#) = accept$.
- Si I contiene $A \to \alpha\bullet$ donde $A \to \alpha$ es una regla enumerada bajo k entonces para cada terminal Y [2], $M(I, Y) = rk$ para denotar un reduce usando la regla número k.

Ejemplo 2 (Gramática de expresiones aritméticas sencilla)**.**
El ejemplo de la gramática de expresiones aritméticas sencilla se puede consultar completo en el archivo ejemploParLR0.pdf disponible en la sección de notas del curso.
La tabla final en ese archivo está calculada usando la técnica SLR que revisaremos a continuación.

SLR

El análisis simple bottom-up o *simple LR* utiliza el autómata construido para LR(0) pero con la tabla de parsing ligeramente diferente como se muestra en el siguiente algoritmo que adicionalmente usa la función auxiliar Follow.

Input: Una gramática aumentada.

Output: La tabla de parsing SLR con las funciones Action and GoTo para la gramática aumentada.

S es el nuevo símbolo inicial y E es el símbolo inicial de la gramática original.

1. Construir la colección de conjuntos de items según el método LR(0): $C = \{I_0, I_1, \ldots, I_n\}$
2. El estado i se construye desde I_i y las acciones están determinadas como sigue:
   - Si el item $A \to \alpha \bullet a\beta$ está en I_i y $\text{GoTo}(I_i, a) = I_j$ entonces $\text{Action}(i, a) =$ shift j donde a es un símbolo terminal.
   - Si el item $A \to \alpha\bullet$ está en I_i entonces $\text{Action}(i, a) =$ reduce$A \to \alpha$ para toda a en $\text{Follow}(A)$ donde A no es el nuevo símbolo de inicio en la gramática aumentada.

[2] En el caso de LR(0), se pueden usar los $Y \in \text{Follow}(A)$.

- Si el item $S \to E\bullet$ está en I_i entonces $\text{ACTION}(i, a) = \text{accept}$.

3. Las transiciones GOTO para un estado i están construidas para todos los símbolos no-terminales de la gramática aumentada.
4. Todas las entradas no definidas por los pasos 2. y 3. son un error.
5. El estado inicial del parser está construido por el conjunto de items que contiene a $S \to \bullet E$.

Un analizador LR(0) tiene conflictos si para una entrada de la tabla se pueden realizar dos acciones diferentes. Un analizador SLR puede construir una tabla con más entradas y posiblemente eliminar conflictos si se lee por adelantado uno o más símbolos de la cadena de entrada. En general, al incluir símbolos por adelantado (look-aheads) se mejora la toma de decisiones.

A continuación veremos dos extensiones de **LR** con *lookahead* de un símbolo. Ambos métodos consideran items extendidos, es decir que tienen dos componentes, la izquierda que son los items de la misma forma que los métodos anteriores (producciones con un punto o bandera, eg. $A \to \alpha \bullet \beta$) y la parte derecha que es el símbolo lookahead: $[A \to \alpha \bullet \beta, \quad b]$ donde b es el look-ahead (un símbolo terminal o el símbolo de final de cadena #).

Un item $[A \to \alpha \bullet \beta, \quad b]$ indica que ya se ha reconocido a α y se espera encontrar a βb en la cadena de entrada.

LR(1)

Para este método se modifican las funciones CLOSURE y GOTO:

Definición 4 (Closure en LR(1))**.** Sea I un conjunto de items, para cada elemento del conjunto de la forma $[A \to \alpha \bullet B\beta, \quad a]$ y por cada producción $B \to \gamma$ agregar a I

$[B \to \bullet\gamma, \quad \text{FIRST}(\beta a)]$

$[B \to \bullet\gamma, \quad \varepsilon]$ si $\text{FIRST}(\beta a) = \varnothing$

```
repeat{
   for each item [A → α • Bβ,a] in I
       for each grammar production B → γ
          for each terminal symbol b in FIRST(βa)
                    add [B → γ,b] to I
}
until no more items are added to I
```

Definición 5 (Goto en LR(1))**.** Dado un conjunto de items I y X un símbolo cualquiera de la gramática, la función GOTO del estado I mediante X se calcula con el siguiente algoritmo:

```
initialize J to be the empty set;
for each item [A → α • Xβ,a] in I
    add [A → αX • β,b] to J
return CLOSURE(J)
```

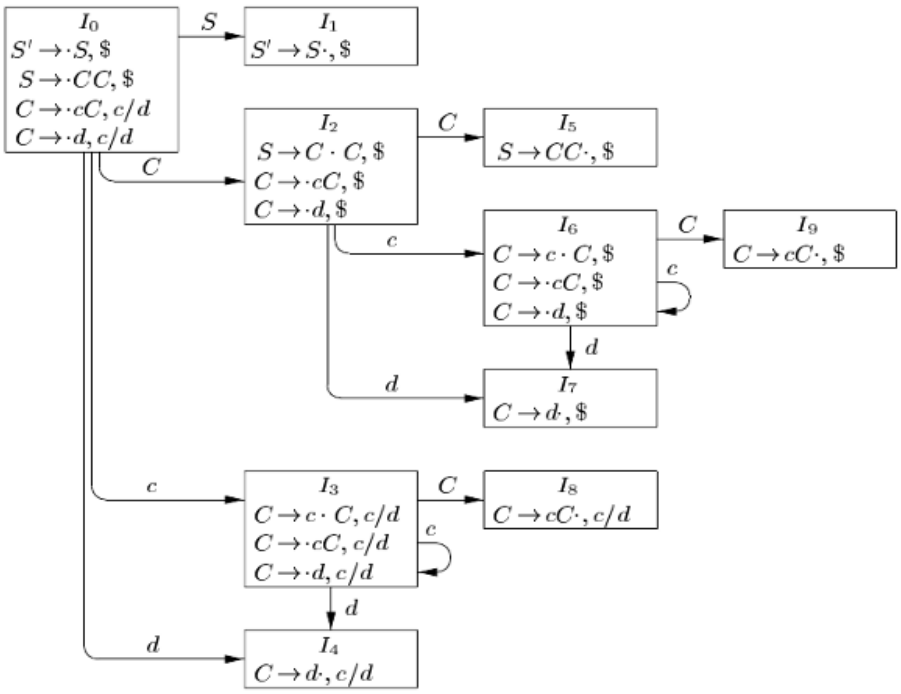
El algoritmo que construye los conjuntos de items `C` en LR(1) para una gramática extendida con el símbolo S es el siguiente:

```
initialize C to { CLOSURE({[S → • E,#]});
repeat{
   for each set of items I in C
       for each grammar symbol X
          if (GOTO(I,X) is not empty and not in C)
          then add GOTO(I,X) to C
}
until no new sets are added to C
```

El algoritmo para llenar la tabla es semejante al que se describió para LR(0):
dado un estado I_i, la acción para ese estado está determinada por:

- Si $[A \to \alpha \bullet a\beta,\ \ b]$ está en I_i con a un terminal y $\text{GoTo}(I_i, a) = I_j$ entonces la acción en $M(I_i, a) = sJ$ para denotar un shift de a y transitar al estado J.
- Si $[A \to \alpha\bullet,\ \ b]$ está en I_i con $A \neq S$ del nuevo inicial, entonces la acción en $M(I_i, b) = rk$ para denotar un reduce usando la regla número k.
- $[S \to E\bullet, \#]$ está en I_i entonces la acción en $M(I_i, \#) = accept$.

Ejemplo 3. Ejemplo 4.54 tomado de [1].
Considere la gramática $S' \to S$ $S \to CC$ $C \to cC \mid d$
El siguiente es el diagrama de los estados y las transiciones de la gramática siguiendo el método LR(1).



La tabla obtenida del autómata anterior es:

State	ACTION			GOTO	
	c	d	\$	S	C
0	s3	s4		1	2
1			acc		
2	s6	s7			5
3	s3	s4			8
4	r3	r3			
5			r1		
6	s6	s7			9
7			r3		
8	r2	r2			
9			r2		

En ambas imágenes el símbolo \$ denota el fin de cadena o el símbolo que hemos usado para esto #.

LALR

Este método es el más usado en la práctica ya que las tablas de parsing son más pequeñas. La idea de este método es hacer más compacto el autómata al unir estados que tienen el mismo núcleo o *core*. Esta idea parte de la observación en que las acciones *shift* dependen sólo del núcleo de un estado. Esta unión de estados no puede producir conflictos de tipo shift/reduce si la gramática ya era de clase LR(1).

Definición 6 (Core de un estado)**.** El núcleo o core de un estado es el conjunto de items donde la componente izquierda es la misma, es decir los lookaheads pueden ser diferentes.

Para este método:

1. Los estados del autómata finito son conjuntos (canónicos) de items usando la función Closure() y comenzando por la cerradura del item $[S' \to \bullet S, \#]$
2. Las transiciones entre estados están etiquetadas por símbolos y calculadas por la función GoTo
3. Los lookaheads son elementos de

Veamos el algoritmo:

Input: Una gramática aumentada con un nuevo símbolo inicial S'

Output: La tabla de parsing LALR con las funciones Action and GoTo para la gramática aumentada.

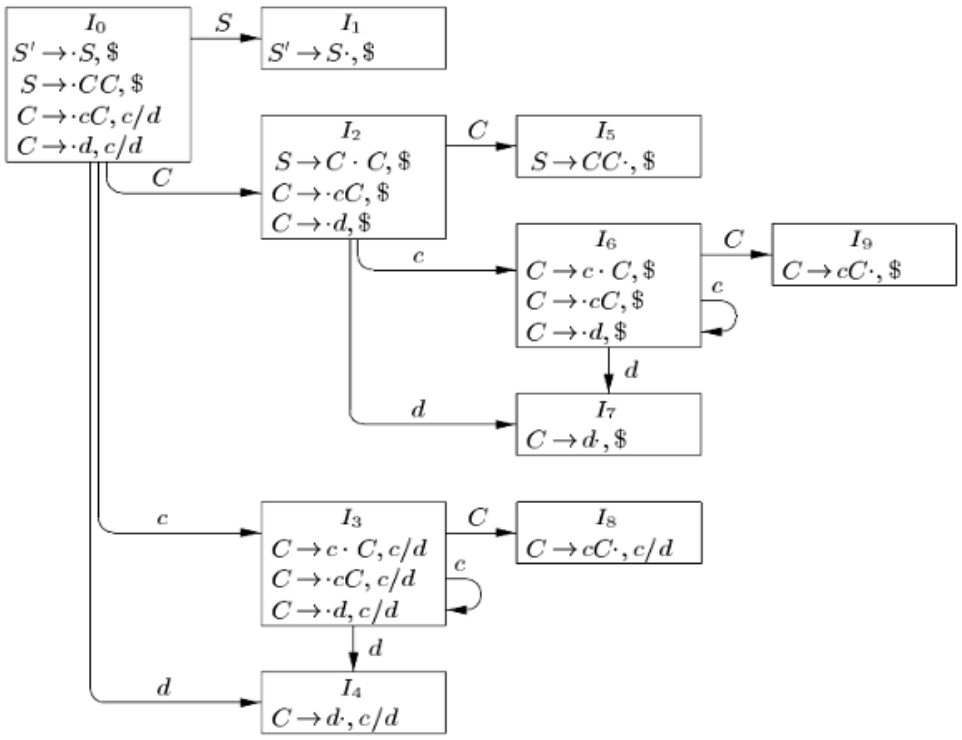
1. Construir la colección de conjuntos de items según el método LR(1) $C = \{I_0, I_1, \ldots, I_n\}$.

2. Para cada núcleo en los conjuntos en C, encontrar todos los conjuntos que compartan el mismo núcleo y reemplazarlos por la unión de ellos.
3. Sea $C' = \{J_0, J_1, \ldots, J_m\}$ la coleción resultante del paso 2.
   Las acciones para el estado j están construidas desde J_j en la misma forma que el método para LR(1) o SLR .
   Si hay un conflicto entonces el algoritmo falla y la gramática no pertenece a la clase LALR(1).
4. La parte de la tabla GoTo se construye de la siguiente forma:
   Si J es la unión de uno o más conjuntos de items según el método LR(1), $J = I_0 \bigcap I_1 \bigcap \ldots \bigcap I_k$[3] entonces los cores de $\text{GoTo}(I_1, X)$, $\text{GoTo}(I_2, X), \ldots,$ $\text{GoTo}(I_k, X)$, serán el mismo dado que comparten las componentes izquierdas. Entonces K será la unión de los conjuntos de items que tienen el mismo core que $\text{GoTo}(I_1, X)$ haciendo $K = \text{GoTo}(J, X)$.

Ejemplo 4. Considere la gramática

$$S' \to S \qquad S \to CC \qquad C \to cC \mid d$$

y el autómata construido con el método **LR(1)**:



La tabla compacta siguiendo el método anterior es:

[3] J es la intersección de los estados y dará como resultado el núcleo compartido.

State	ACTION			GOTO	
	c	d	\$	S	C
0	s36	s47		1	2
1			acc		
2	s36	s47			5
36	s36	s47			89
47	r3	r3	r3		
5			r1		
89	r2	r2	r2		

Construcción eficiente de la tabla para LALR A pesar de que el método recién descrito para compactar la tabla de parsing es correcto, se puede hacer más eficiente el cálculo de los items. Veremos una forma de mejorar la construcción de la tabla para la clase **LALR** en donde se evitará construir todos los conjuntos de items que se obtienen por el método visto para **LR(1)**. Esto se realizará al seguir el método de **LR(0)** y calcular los lookaheads ya sea espontáneamente o propagarlos.

1. Primero es necesario calcular los conjuntos de items según el método de **LR(0)**, es decir mediante conjuntos canónicos usando la función Closure para los estados e iniciando con $[S \to \bullet E, \#]$ y la función GoTo para las transiciones siguiendo las definiciones 2 y 3.

2. Después se deben identificar los items que sean tipo kernel para cada estado, siguiendo la definición 1 (item de la producción inicial $S \to \bullet E$ donde E es el símbolo inicial de la gramática original o cualquier item que no tenga la bandera más a la izquierda).

3. Completar los items con los lookaheads adecuados y así generar los kernels de LALR.
   Para esto existen dos formas:

   a) Lookaheads espontáneos
   Sea I un conjunto de items con kernel $[A \to \alpha \bullet \beta, a]$ y $J = \text{GoTo}(I, X)$
   para cualquier símbolo de la gramática X, b será un lookahead espontáneo donde $\text{GoTo}(\text{Closure}(\{[A \to \alpha \bullet \beta, a]\}), X)$ contiene al item $[B \to \gamma \bullet \delta, b]$
   El lookahead # en $[S \to \bullet E, \#]$ se considera como espontáneo.

   b) Lookaheads propagados
   Considerando las mismas condiciones del inciso anterior se toma $a = b$ y $\text{GoTo}(\text{Closure}(\{[A \to \alpha \bullet \beta, b]\}), X)$ contiene al item $[B \to \gamma \bullet \delta, b]$, es decir se propaga el lookahead de $A \to \alpha \bullet \beta$ en el kernel del estado I, hacia $B \to \gamma \bullet \delta$ que está en el kernel del estado J.

Veamos un algortimo para determinar los lookaheads:

Input: El kernel K de un conjunto de items de LR(0) y X un símbolo cualquiera de la gramática.

Output: Los lookaheads generados espontáneamente o los propagados por los items en I para los items del kernel en $J = \text{GoTo}(I, X)$.

Este algoritmo incluye un nuevo símbolo $\diamond$ que no está en la gramática y que servirá para indicar los lookaheads que se propagarán.

```
for each item A→ α • β in K
{
  J:= CLOSURE({[A→ α • β,◇]});
  if [B→ γ • Xδ, a] is in J and a is not ◇
    then lookahead a is generated spontaneously for item B→ γX • δ in GoTo(I,X);
  if [B→ γ • Xδ, a] is in J
    then lookaheads are propagated from item A→ α • β to B→ γX • δ in GoTo(I,X);
}
```

Los lookaheads espontáneos se pueden calcular usando la definición 4.

Ejemplo 5 (LALR con tabla eficiente)**.** Considera la gramática extendida

$$S' \to S \qquad S \to L = R \mid R \qquad L \to \star R \mid \mathsf{id} \qquad R \to L$$

1. Calcular los estados o conjuntos de items usando el método LR(0)

I_0: $S' \to \cdot S$, $S \to \cdot L = R$, $S \to \cdot R$, $L \to \cdot * R$, $L \to \cdot \mathbf{id}$, $R \to \cdot L$

I_1: $S' \to S\cdot$

I_2: $S \to L\cdot = R$, $R \to L\cdot$

I_3: $S \to R\cdot$

I_4: $L \to *\cdot R$, $R \to \cdot L$, $L \to \cdot * R$, $L \to \cdot \mathbf{id}$

I_5: $L \to \mathbf{id}\cdot$

I_6: $S \to L = \cdot R$, $R \to \cdot L$, $L \to \cdot * R$, $L \to \cdot \mathbf{id}$

I_7: $L \to *R\cdot$

I_8: $R \to L\cdot$

I_9: $S \to L = R\cdot$

2. Identificar los kernels de cada conjunto (item inicial o items que no tienen la bandera al inicio de la producción). La siguiente figura muestra los conjuntos de items siguiendo el método LR(0) y están marcados los items kernel de cada uno.

I_0: $S' \to \cdot S$

I_1: $S' \to S\cdot$

I_2: $S \to L\cdot = R$
$R \to L\cdot$

I_3: $S \to R\cdot$

I_4: $L \to *\cdot R$

I_5: $L \to \mathbf{id}\cdot$

I_6: $S \to L = \cdot R$

I_7: $L \to *R\cdot$

I_8: $R \to L\cdot$

I_9: $S \to L = R\cdot$

3. Agregar los lookaheads a los kernels encontrados

   La cerradura del kernel para I_0 es:

   $$[S' \to \bullet S, \diamond] \quad [L \to \bullet * R, \diamond] \quad [S \to \bullet L = R, \diamond] \quad [L \to \bullet * R, =] \quad [S \to \bullet R, \diamond]$$
   $$[L \to \bullet id, \diamond] \quad [R \to \bullet L, \diamond] \quad [L \to \bullet id, =]$$

   aquí se generan espontáneamente los lookaheads = dado que este símbolo pertenece al conjunto FIRST$(= R\diamond)$ siguiendo la definición 4.

4. Propagar los lookaheads iterando el algoritmo anterior

FROM		TO	
I_0:	$S' \to \cdot S$	I_1:	$S' \to S\cdot$
		I_2:	$S \to L\cdot = R$
		I_2:	$R \to L\cdot$
		I_3:	$S \to R\cdot$
		I_4:	$L \to *\cdot R$
		I_5:	$L \to \mathbf{id}\cdot$
I_2:	$S \to L\cdot = R$	I_6:	$S \to L = \cdot R$
I_4:	$L \to *\cdot R$	I_4:	$L \to *\cdot R$
		I_5:	$L \to \mathbf{id}\cdot$
		I_7:	$L \to *R\cdot$
		I_8:	$R \to L\cdot$
I_6:	$S \to L = \cdot R$	I_4:	$L \to *\cdot R$
		I_5:	$L \to \mathbf{id}\cdot$
		I_8:	$R \to L\cdot$
		I_9:	$S \to L = R\cdot$

Ejercicios

1. Considera la siguiente gramática

   $S \to E$

$E \to Aa$

$A \to BC \mid BCf$

$B \to b$

$C \to c$

a) Construye los conjuntos canónicos de items **LR(0)**.

b) Muestra la tabla de parsing para las acciones y la función GoTo.

2. Proporciona el autómata para construir la tabla de parsing **LR(1)** para la siguiente gramática:

   $S \to A$

   $A \to bB$

   $B \to cC$

   $B \to cCe$

   $C \to dA$

   $A \to a$

   Además analiza una cadena no trivial de longitud al menos 4, mostrando la secuencia de acciones del autómata.

3. Muestra que la siguiente gramática G pertenece a la clase **LL(1)** pero no a la clase **SLR(0)**.

   $S \to AaAb \mid BbBa$

   $A \to \varepsilon$

   $B \to \varepsilon$

Referencias

[1] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers, Principles, Techniques and Tools*. Pearson Education Inc., Second edition, 2007.

[2] Jean-Christophe Filliâtre. Curso Compilation (inf564) école Polytechnique, Palaiseau, Francia. `http://www.enseignement.polytechnique.fr/informatique/INF564/`, 2018. Material en francés.

[3] François Pottier. Presentaciones del curso Compilation (inf564) École Polytechnique, Palaiseau, Francia. `http://gallium.inria.fr/~fpottier/X/INF564/`, 2016. Material en francés.

[4] Michael Lee Scott. *Programming Language Pragmatics*. Morgan-Kauffman Publishers, Third edition, 2009.

[5] Yunlin Su and Song Y. Yan. *Principles of Compilers, A New Approach to Compilers Including the Algebraic Method*. Springer-Verlag, Berlin Heidelberg, 2011.

[6] Tim Teitelbaum. Introduction to compilers. `http://www.cs.cornell.edu/courses/cs412/2008sp/`, 2008.

[7] Steve Zdancewic. Notas del curso (CIS 341) - Compilers, Universidad de Pennsylvania, Estados Unidos. https://www.cis.upenn.edu/~cis341/current/, 2018.