Análisis Sintáctico: *Parser bottom-up*
Compiladores 2020-1, Nota de clase 5

Lourdes Del Carmen González Huesca

26 de septiembre de 2019
Facultad de Ciencias UNAM

Los analizadores *bottom-up* construyen un árbol de sintaxis concreta 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 llama shift-reduce donde la reducción se realiza para obtener un proceso inverso de una derivación paso a paso. La clase de gramáticas de estos analizadores son las LR (lectura left-to-right de la cadena de entrada y usando derivaciones más a la derecha). Veremos varias clases de estas gramáticas y la construcción de las máquinas para ellas.

Preliminares

Las gramáticas que consideramos no son ambiguas, para asegurar que existe una única derivación, en este caso reescribiendo el símbolo no-terminal más a la derecha en el proceso.

El análisis **LR** funciona en general de la siguiente forma:

- leer la gramática como un autómata no determinista donde el estado indica una producción parcialmente reconocida y la pila contiene a los estados precedentes;
- transformar este autómata, de ser posible, en uno determinista.

La pila de estados representa la historia de visita de éstos, el tope de la pila es justo el estado actual. Además la entrada es una secuencia de lexemas. En un estado q se pueden realizar dos acciones:

- 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; 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.

Dada una cadena $\alpha\beta w$ (con w cadena de símbolos terminales), se define el **handle** o mango de ella 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 (#), como de la gramática al incluir un nuevo símbolo inicial (S). Las configuraciones del autómata cambian respecto a cada método pero la inicial debe tener por un lado el símbolo de fondo de la pila 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 S y por otro lado se consume la cadena de entrada y sólo resta el símbolo de fin de cadena.

Ventajas del análisis **LR**:

- 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

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

Los analizadores **LR(k)** reconocen las cadenas que puedan estar a la derecha de una producción y después se dedice cuál es la producción indicada, de esta forma se revisan varias posibilidades de la 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.

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)**.

Algoritmo para parser LR

El algoritmo se sirve de la cadena de entrada y una tabla de parsing que contiene las acciones (shift, reduce o accept) y las transiciones entre estados:

Input: Cadena de entrada w y la tabla LR con las funciones ACTION and 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, lo que estudiaremos son variantes en la construcción de la tabla y del autómata que guarda el avance en el análisis de la cadena de entrada.

LR(0)

Para este método, los estados serán conjuntos de producciones de la gramática donde se ha identificado una subcadena:

Definición 1 (Item)**.** Un item es una producción de la gramática con un punto ($\bullet$) 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 puntno más a la izquierda excepto el item inicial $S \to \bullet E$

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:

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 $A \to \alpha X \bullet \beta$ está en I.

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

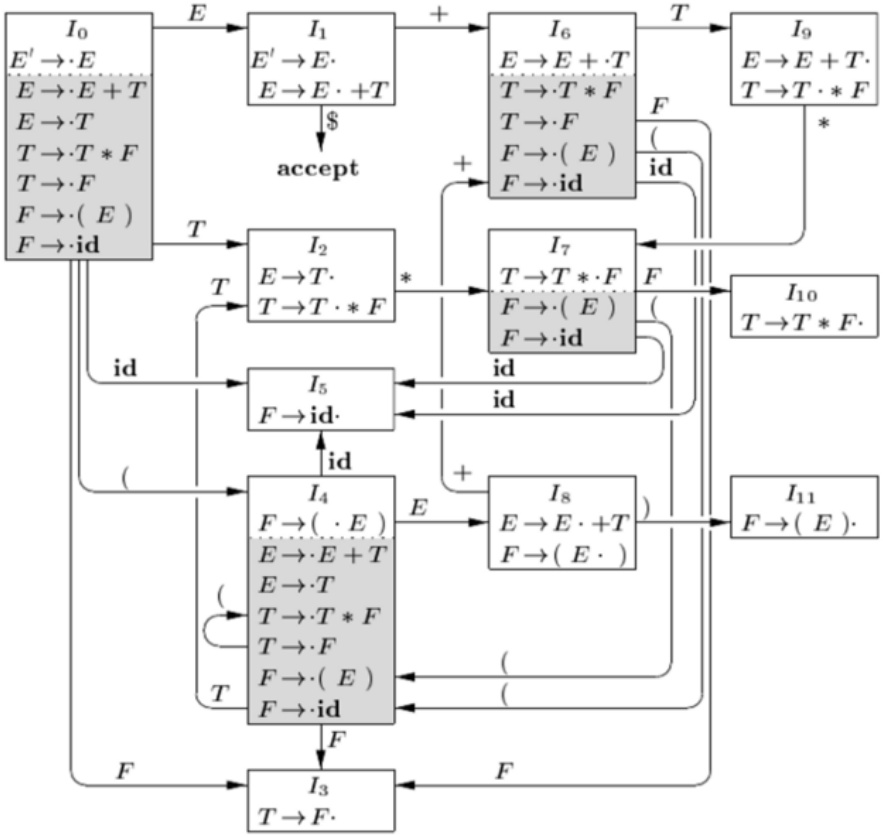
```
C = CLOSURE({S → • E});
repeat{
   for each set of items I in C
       for each grammar symbol X
          if (GOTO(I,X) is not empty and not in C)
          then add GOTO(I,X) to C
}
until no new sets are added to C
```

Ejemplo 1 (Gramática de expresiones aritméticas sencilla)**.**
El siguiente ejemplo es la tabla de parsing LR(0) para la gramática de expresiones aritméticas sencilla donde cada producción está numerada, la producción inicial (0) en la gramática extendida es $E' \to E$.

(1) $E \to E + T$
(2) $E \to T$
(3) $T \to T * F$
(4) $T \to F$
(5) $F \to (E)$
(6) $F \to \mathbf{id}$

STATE	ACTION						GOTO		
	id	+	*	(	)	$	E	T	F
0	s5			s4			1	2	3
1		s6				acc			
2		r2	s7		r2	r2			
3		r4	r4		r4	r4			
4	s5			s4			8	2	3
5		r6	r6		r6	r6			
6	s5			s4				9	3
7	s5			s4					10
8		s6			s11				
9		r1	s7		r1	r1			
10		r3	r3		r3	r3			
11		r5	r5		r5	r5			

El autómata para esta gramática es:



Donde la siguiente tabla es el análisis de la cadena **id** ⋆ **id**

LINE	STACK	SYMBOLS	INPUT	ACTION
(1)	0	\$	**id** $*$ **id** \$	shift to 5
(2)	0 5	\$ **id**	$*$ **id** \$	reduce by $F \to$ **id**
(3)	0 3	\$ F	$*$ **id** \$	reduce by $T \to F$
(4)	0 2	\$ T	$*$ **id** \$	shift to 7
(5)	0 2 7	\$ $T*$	**id** \$	shift to 5
(6)	0 2 7 5	\$ $T*$ **id**	\$	reduce by $F \to$ **id**
(7)	0 2 7 10	\$ $T*F$	\$	reduce by $T \to T*F$
(8)	0 2	\$ T	\$	reduce by $E \to T$
(9)	0 1	\$ E	\$	accept

SLR

El análisis simple o simple LR es utilizar el autómata construido para LR(0) pero la tabla de parsing es ligeramente diferente como se muestra en el siguiente algoritmo:

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

Ejemplo 2. Considera la gramática

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

Los conjuntos de items están mostrados en el último ejemplo de esta nota, si se calculan las acciones correspondientes a cada uno de ellos se puede ver que en el estado dos hay dos acciones para el símbolo =:

- shift y avanzar al estado I_6
- reduce con la producción $R \to L$

Veremos que este conflicto se resuelve mediante otro método.

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, eg. $A \to \alpha \bullet \beta$) y la parte derecha que es el símbolo lookahead: $[A \to \alpha \bullet \beta, a]$ donde a es un símbolo terminal o el símbolo de final de cadena #.

LR(1)

Para este método se modifican las funciones CLOSURE y GOTO:

Definición 4 (Closure en LR(1))**.** Sea I un conjunto de items, la cerradura de I o CLOSURE(I) son los items calculados con el algoritmo siguiente:

```
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 un símbolo cualquiera de la gramática X, 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)
```

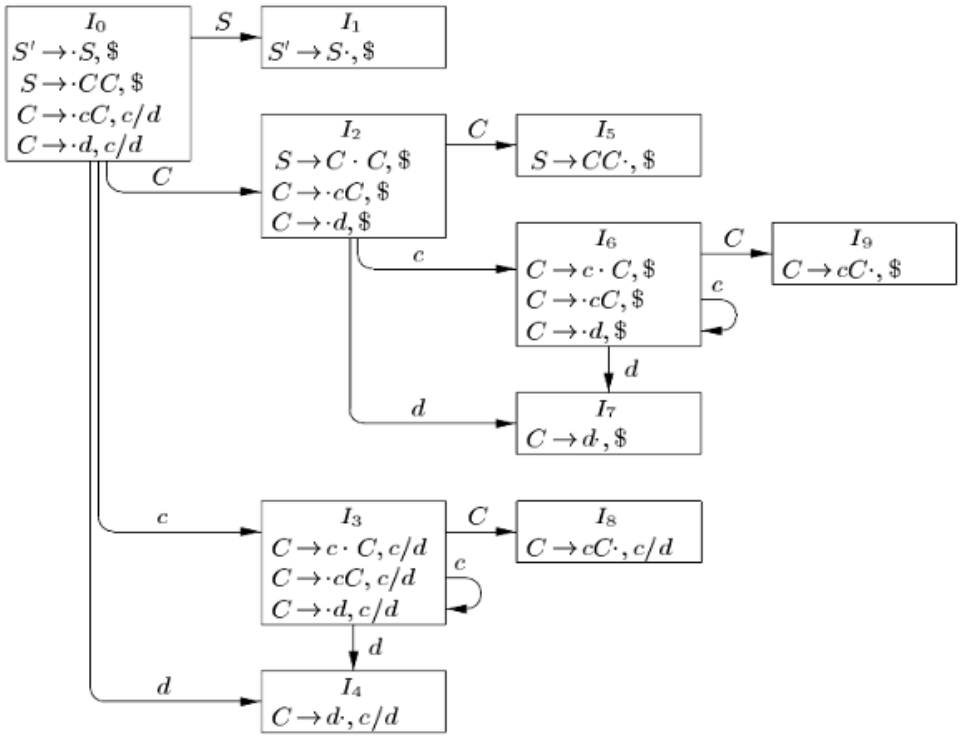
El algoritmo que construye los conjuntos de items C en LR(1) para una gramática extendida con el símbolo S es el siguiente:

```
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
```

Ejemplo 3. Considere la gramática

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

La siguiente figura muestra los estados y las transiciones de la gramática siguiendo el método LR(1).



LRLA

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.

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.

Esta unión de estados no puede producir conflictos de tipo shift/reduce si la gramática ya era de clase LR(1). Veamos el algoritmo:

Input: Una gramática aumentada.

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 de 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 colleción 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) (ver algoritmo . Si hay un conflicto entonces el algoritmo falla y la gramática no pertenece a la clase LALR(1).
4. La tabla GOTO se construye de la siguiente forma:
   Si J es la unión de núcleos de uno o mś conjuntos de items, $J = I_0 \cup I_1 \cup \cdots \cup I_k$ entonces los cores de $\text{GOTO}(I_1, X)$, $\text{GOTO}(I_2, X), \ldots, \text{GOTO}(I_k, X)$, son el mismo dado que comparten las componentes izquierdas. Entonces K será la unión de los conjuntos d eitems que tienen el mismo core que $\text{GOTO}(I_1, X)$ y es justo $K = \text{GOTO}(J, X)$.

Construcción eficiente de la tabla para LALR Veremos una forma de mejorar la construcción de la tabla para la clase **LALR** en donde se evitá 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 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.

3. Completar los items con los lookaheads apropiados 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 $[B \to \gamma \bullet \delta, b]$

b) Lookaheads propagados
Considerando las mismas condiciones del inciso anterior se toma $a = b$ y $\textsc{GoTo}(\textsc{Closure}(\{[A \to \alpha \bullet \beta, b]\}), X)$ 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 $J = \textsc{GoTo}(I, X)$.

Este algoritmo incluye un nuevo símbolo $\diamond$, 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);
}
```

Ejemplo 4 (Lookaheads)**.** Considera la gramática extendida

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

La cerradura del kernel para I_0 es:

$$\begin{array}{lll} S' \to & \bullet S, \diamond & L \to \bullet * R, \diamond \\ S \to & \bullet L = R, \diamond & L \to \bullet * R, = \\ S \to & \bullet R, \diamond & L \to \bullet id, \diamond \\ R \to & \bullet L, \diamond & L \to \bullet id, = \end{array}$$

La siguiente figura muestra los conjuntos de items siguiendo el método LR(0) y están marcados los items kernel de cada uno.

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

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: $S \to L = R\cdot$ ★

La siguiente tabla muestra los items que propagan lookaheads:

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:	$S \to L = R\cdot$

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] Frank Pfenning. Notas del curso (15-411) Compiler Design, 2014. `https://www.cs.cmu.edu/~fp/courses/15411-f14/`.

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

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

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

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