Procesadores de Lenguajes

Ingeniería Técnica superior de Ingeniería Informática

Departamento de Lenguajes y Sistemas informáticos

5

Análisis sintáctico

Analizadores ascendentes

Javier Vélez Reyes
jvelez@lsi.uned.es
Departamento de Lenguajes Y Sistemas Informáticos
UNED

UNED

ETS de Ingeniería Informática

Objetivos

- Comprender el análisis ascendente por reducción desplazamiento
- Aprender el algoritmo general de análisis ascendente
- Conocer la arquitectura general de un analizador sintáctico ascendente
- Aprender a construir analizadores ascendentes en complejidad creciente
  - Analizadores LR (0)
  - Analizadores SLR
  - Analizadores LALR
  - Analizadores LR (1)
- Entender cómo los analizadores sintácticos realizan la gestión de errores
  - Entender los conflictos reducción – reducción
  - Entender los conflictos reducción – desplazamiento
- Entender la clasificación de gramáticas inducida en virtud de los métodos de análisis empleados

Índice

Índice

¿Qué es el análisis ascendente?

El análisis sintáctico ascendente es una técnica de análisis sintáctico que intenta comprobar si una cadena x pertenece al lenguaje definido por una gramática L (G) aplicando los siguientes criterios

- Partir de los elementos terminales de la frase x
- Escoger reglas gramaticales estratégicamente
- Aplicar a la inversa derivaciones por la derecha (Right Most Derivation)
- Procesar la cadena de izquierda a derecha
- Intentar alcanzar el axioma para obtener el árbol de análisis sintáctico o error

R_1: E ::= E + E

R_2: E ::= E - E

R_3: E ::= E * E

R_4: E ::= E / E

R_5: E ::= (E)

R_6: E ::= n

Cadena de derivación

El análisis ascendente genera una cadena de derivación por la derecha leída en sentido inverso

n + n * n ←

E + n * n ←

E + E * n ←

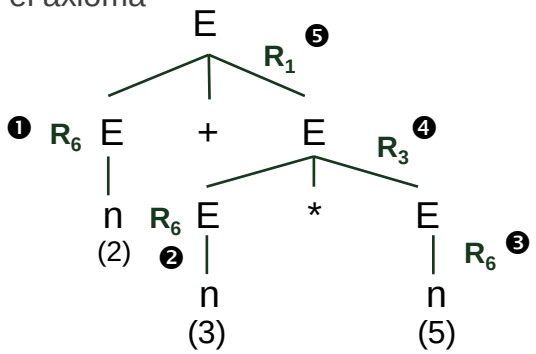
E + E * E ←

E + E ←

E

Árbol de análisis sintáctico

Se parte la cadena de entrada leída de izquierda a derecha y se aplican reglas a la inversa para intentar alcanzar el axioma



Analizadores sintácticos ascendentes

¿Como se selecciona el siguiente no terminal a derivar?

¿Como se selecciona la regla de producción en cada paso de derivación inversa?

Analizadores deterministas

Son analizadores que imponen ciertas restricciones sobre la naturaleza de los lenguajes que soportan

Analizadores descendentes

Se parte del axioma y se aplica una cadena de derivaciones para construir un árbol sintáctico

Analizadores ascendentes

Se parte de los terminales y se construye la inversa de una derivación para intentar alcanzar el axioma

LR (0)

Determinan la regla que hay que aplicar sin consultar terminales a la entrada

SLR

LALR

LR (1)

Determinan la regla a aplicar consultado el primer terminal a la entrada

LR (k)

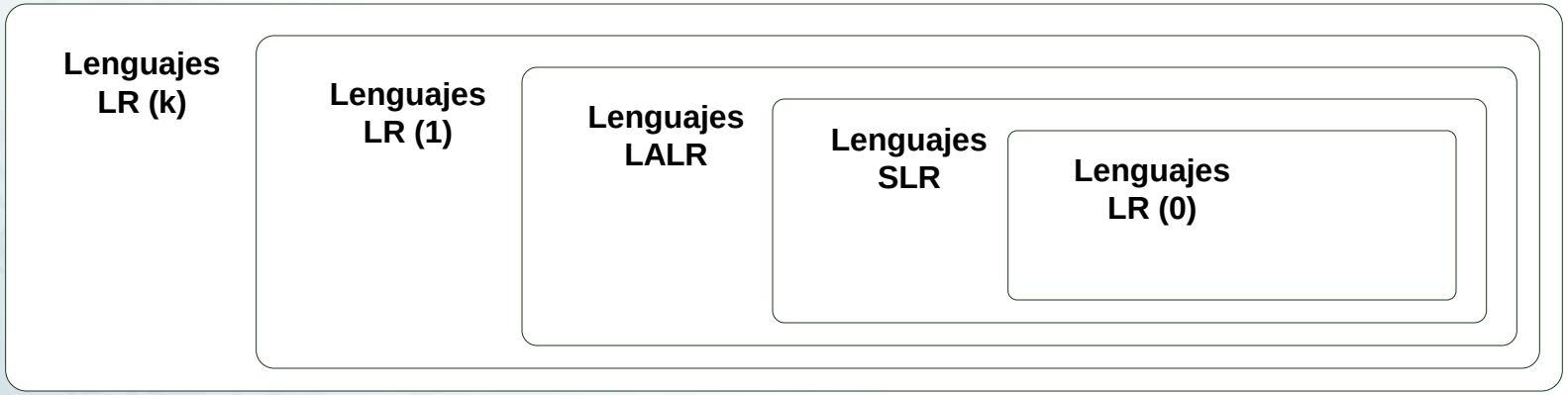
Determinan la regla a aplicar consultando los k primeros terminales a la entrada

Análisis sintáctico ascendente

Los analizadores sintácticos ascendentes son capaces de decidir qué regla de producción aplicar a cada paso en función de los elementos terminales que se encuentran en la cabeza de lectura de la cadena de entrada. Como consecuencia se consigue un proceso de análisis con complejidad lineal O (n) con respecto al tamaño del problema. Estos analizadores son llamados genéricamente analizadores LR (K) o analizadores por reducción desplazamiento

- Lectura de izquierda a derecha
- Derivación más a la derecha (Right Most Derivatión)
- Número de terminales consultados para determinar la regla de producción a aplicar

LR (k)



Análisis sintáctico ascendente

Los analizadores sintácticos ascendentes son capaces de decidir qué regla de producción aplicar a cada paso en función de los elementos terminales que se encuentran en la cabeza de lectura de la cadena de entrada. Como consecuencia se consigue un proceso de análisis con complejidad lineal O (n) con respecto al tamaño del problema. Estos analizadores son llamados genéricamente analizadores LR (K) o analizadores por reducción desplazamiento

r_1. E ::= E + E
r_2. E ::= E - E
r_3. E ::= E * E
r_4. E ::= E / E
r_5. E ::= (E)
r_6. E ::= n

Análisis por reducción – desplazamiento

2 operaciones

A cada paso el analizador decide realizar una de dos operaciones posibles

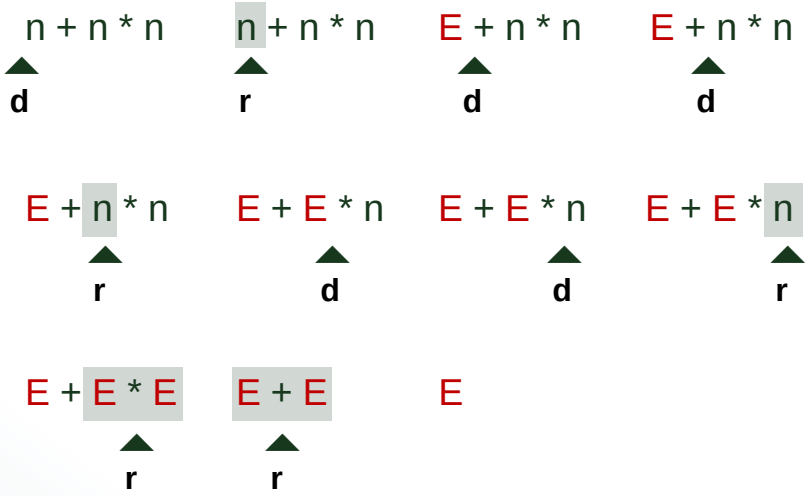
Desplazar

El analizador debe procesar un número suficiente de terminales desplazando la cabeza lectora antes de aplicar una regla de producción a la inversa. Esta operación se llama desplazar

Reducir

Cuando se reconoce en la forma de frase en curso un fragmento igual a la parte derecha de una regla se sustituye éste por el antecedente. Esta operación se llama reducir

mango
d *desplazar*
r *reducir*



Análisis sintáctico ascendente

Los analizadores sintácticos ascendentes son capaces de decidir qué regla de producción aplicar a cada paso en función de los elementos terminales que se encuentran en la cabeza de lectura de la cadena de entrada. Como consecuencia se consigue un proceso de análisis con complejidad lineal O (n) con respecto al tamaño del problema. Estos analizadores son llamados genéricamente analizadores LR (K) o analizadores por reducción desplazamiento

r_1. E ::= E + E
r_2. E ::= E - E
r_3. E ::= E * E
r_4. E ::= E / E
r_5. E ::= (E)
r_6. E ::= n

Análisis por reducción – desplazamiento

2 tipos de conflictos

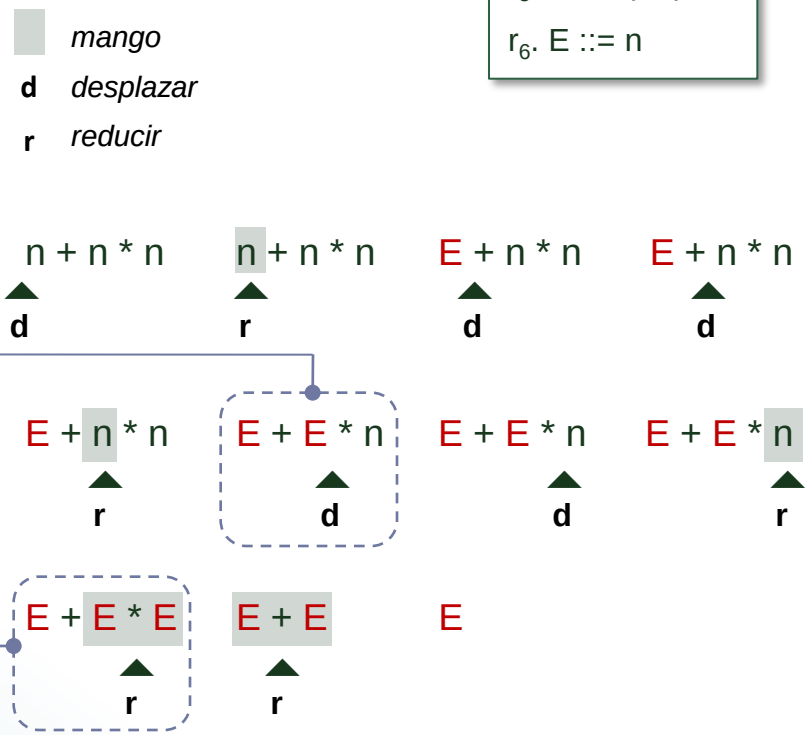
Pueden surgir en este proceso 2 tipos de conflictos que el analizador debe ser capaz de resolver

Reducción - desplazamiento

El analizador puede tanto aplicar un paso de reducción como uno de desplazamiento

Reducción - reducción

El analizador puede aplicar dos reglas distintas para realizar diferentes pasos de reducción

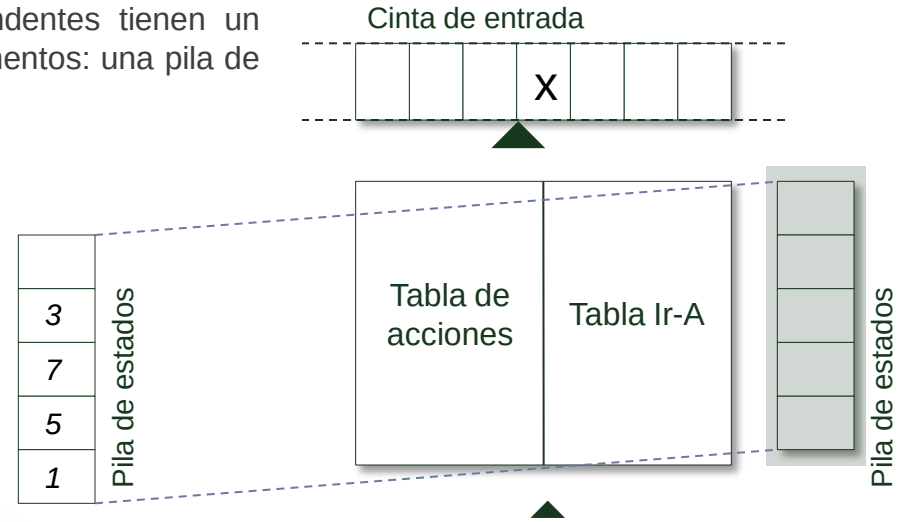


Análisis sintáctico ascendente

Los analizadores sintácticos ascendentes son capaces de decidir qué regla de producción aplicar a cada paso en función de los elementos terminales que se encuentran en la cabeza de lectura de la cadena de entrada. Como consecuencia se consigue un proceso de análisis con complejidad lineal O (n) con respecto al tamaño del problema. Estos analizadores son llamados genéricamente analizadores LR (K) o analizadores por reducción desplazamiento

Arquitectura general de análisis sintáctico ascendente

Todos los tipos de analizadores sintácticos ascendentes tienen un mismo modelo de arquitectura que consta de 3 elementos: una pila de estados, una tabla de acciones y una tabla Ir-A



I. Pila de estados

El analizador sintáctico puede atravesar, a lo largo del proceso de análisis, distintos estadios de compilación que dependen del lenguaje analizado. Estos estadios se identifican con estados codificados numéricamente que se gestionan en una pila. En cada momento el estado en curso aparece en la cima de la pila

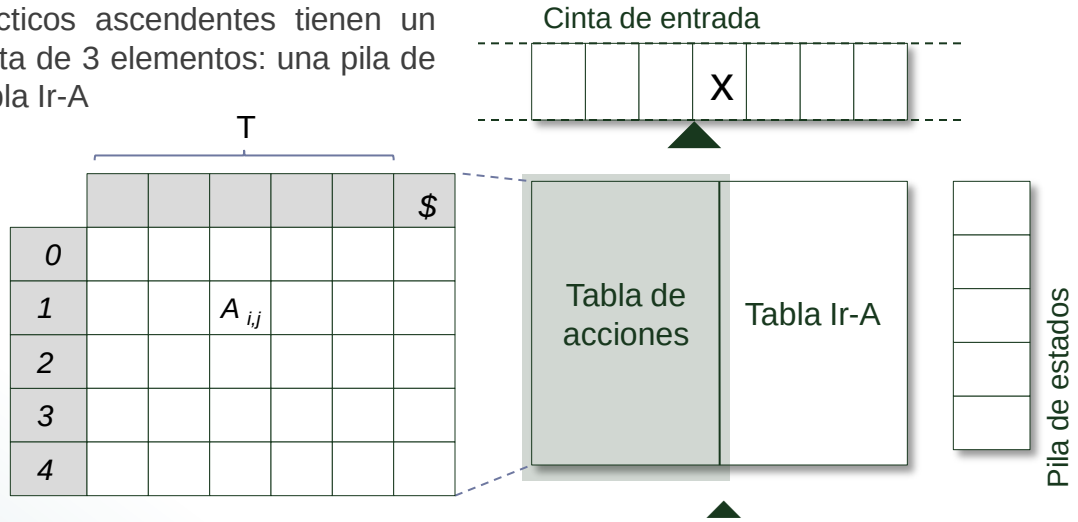
La diferencia entre los tipos de analizadores sintácticos estriba en cómo se construyen estas tablas y en el tipo de gramáticas (lenguajes) que admiten

Análisis sintáctico ascendente

Los analizadores sintácticos ascendentes son capaces de decidir qué regla de producción aplicar a cada paso en función de los elementos terminales que se encuentran en la cabeza de lectura de la cadena de entrada. Como consecuencia se consigue un proceso de análisis con complejidad lineal O (n) con respecto al tamaño del problema. Estos analizadores son llamados genéricamente analizadores LR (K) o analizadores por reducción desplazamiento

Arquitectura general de análisis sintáctico ascendente

Todos los tipos de analizadores sintácticos ascendentes tienen un mismo modelo de arquitectura que consta de 3 elementos: una pila de estados, una tabla de acciones y una tabla Ir-A



II. Tabla de acciones

Para cada estado y cada terminal (o $) a la entrada el analizador tiene información de que acciones debe realizar. Las operaciones posibles son:

- d_j Desplazar y apilar el estado j
- r_k Reducir por la regla r
- e Emitir un error
- Ok Aceptar la cadena de entrada

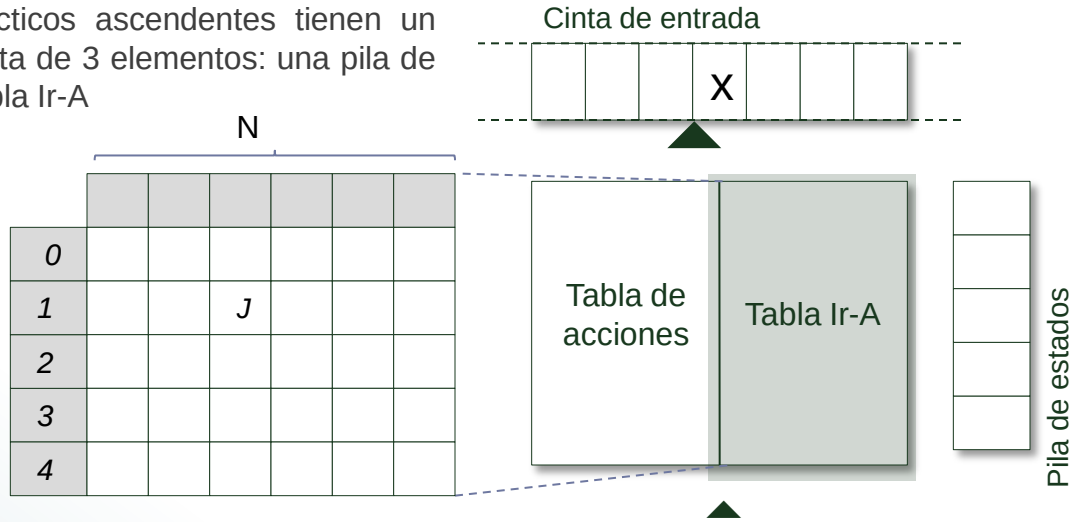
La diferencia entre los tipos de analizadores sintácticos estriba en cómo se construyen estas tablas y en el tipo de gramáticas (lenguajes) que admiten

Análisis sintáctico ascendente

Los analizadores sintácticos ascendentes son capaces de decidir qué regla de producción aplicar a cada paso en función de los elementos terminales que se encuentran en la cabeza de lectura de la cadena de entrada. Como consecuencia se consigue un proceso de análisis con complejidad lineal O (n) con respecto al tamaño del problema. Estos analizadores son llamados genéricamente analizadores LR (K) o analizadores por reducción desplazamiento

Arquitectura general de análisis sintáctico ascendente

Todos los tipos de analizadores sintácticos ascendentes tienen un mismo modelo de arquitectura que consta de 3 elementos: una pila de estados, una tabla de acciones y una tabla Ir-A



III. Tabla Ir-A

En el caso de aplicar reducción, y del estado de compilación en curso, esta tabla indica el estado que hay que apilar en función del antecedente de la regla de producción aplicada

La diferencia entre los tipos de analizadores sintácticos estriba en cómo se construyen estas tablas y en el tipo de gramáticas (lenguajes) que admiten

Análisis sintáctico ascendente

Los analizadores sintácticos ascendentes son capaces de decidir qué regla de producción aplicar a cada paso en función de los elementos terminales que se encuentran en la cabeza de lectura de la cadena de entrada. Como consecuencia se consigue un proceso de análisis con complejidad lineal O (n) con respecto al tamaño del problema. Estos analizadores son llamados genéricamente analizadores LR (K) o analizadores por reducción desplazamiento

Algoritmo general de análisis sintáctico ascendente

El algoritmo de análisis sintáctico ascendente también es independiente del tipo de analizador que se utilice. Una vez que disponemos de la información de las tablas acción e ir-A se procede de la siguiente forma

```
pila = [0]
a = scanner.nextToken ()
do {
    s = pila.cima ()
    if ( Acción [s, a] == dj ) {
         pila.push (j)
         a = scanner.nextToken ()
    }
```

```
    } else if ( acción [s, a] == rk ) {
        for (int i=0; i < longitud (parte derecha rk); i++)
          pila.pop ()
        p = pila.cima ()
        A = ParteIzquierda (rk)
        pila.push (irA (p, A))
    } else if ( Acción (s, a) = aceptar) return
      else throw new SyntaxError ()
} while (true)
```

Análisis sintáctico ascendente

r_0. S ::= S' ::= S

r_1. S ::= B A end

r_2. A ::= begin C

r_3. C ::= codigo

r_4. B ::= tipo

r_5. B ::= id B

Los analizadores sintácticos ascendentes son capaces de decidir qué regla de producción aplicar a cada paso en función de los elementos terminales que se encuentran en la cabeza de lectura de la cadena de entrada. Como consecuencia se consigue un proceso de análisis con complejidad lineal O (n) con respecto al tamaño del problema. Estos analizadores son llamados genéricamente analizadores LR (K) o analizadores por reducción desplazamiento

Algoritmo general de análisis sintáctico ascendente

El algoritmo de análisis sintáctico ascendente también es independiente del tipo de analizador que se utilice. Una vez que disponemos de la información de las tablas acción e ir-A se procede de la siguiente forma

Ejemplo

	end	begin	codigo	tipo	id	$	S	A	B	C
0	-	-	-	d3	d4	-	1		2	
1	-	-	-	-	-	ok				
2	-	d6	-	-	-	-		5		
3	-	r4	-	-	-	-				
4	-	-	-	d3	d4	-			7	
5	d8	-	-	-	-	-				
6	-	-	d10	-	-	-				9
7	-	r5	-	-	-	-				
8	-	-	-	-	-	r1				
9	r2	-	-	-	-	-				
10	r3	-	-	-	-	-				

PILA	ENTRADA	ACCIÓN
0	id tipo begin codigo end $	d4
0 4	tipo begin codigo end $	d3
0 4 3	begin codigo end $	r4
0 4 7	begin codigo end $	r5
0 2	begin codigo end $	d6
0 2 6	codigo end $	d10
0 2 6 10	end $	r3
0 2 6 9	end $	r2
0 2 5	end $	d8
0 2 5 8	$	r1
0 1	$	Ok

Analizadores sintácticos ascendentes

Los diferentes tipos de analizadores sintácticos ascendentes – LR (0), SLR, LALR, LR (1) y LR (k) – sólo se diferencian entre sí por el procedimiento de construcción de las tablas acción e ir-A. En esta sección abordaremos su estudio

Conceptos esenciales

La construcción de las tablas de análisis acción e ir-A para cada uno de los cuatro tipos de analizadores sintácticos utilizan 4 tipos de conceptos. Éstos tienen una interpretación distinta para analizadores LR (0), SLR, LR (1) y LALR (1)

	Elemento	P. Viable	F. Cierre	F. Ir-A	C. Canónica	Autómata
LR (0)	LR (0)	LR (0)	LR (0)	LR (0)	LR (0)	LR (0)
SLR	LR (0)	LR (0)	LR (0)	LR (0)	LR (0)	LR (0)
LALR	LALR	LALR	LR (1)	LR (1)	LALR	LALR
LR (1)	LR (1)	LR (1)	LR (1)	LR (1)	LR (1)	LR (1)

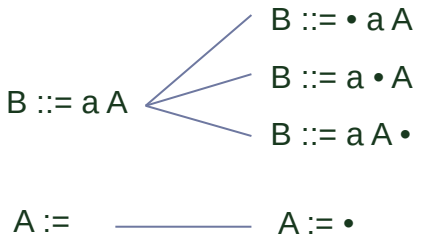
Analizadores sintácticos ascendentes

Los diferentes tipos de analizadores sintácticos ascendentes – LR (0), SLR, LALR, LR (1) y LR (k) – sólo se diferencian entre sí por el procedimiento de construcción de las tablas acción e ir-A. En esta sección abordaremos su estudio

Conceptos esenciales LR (0)

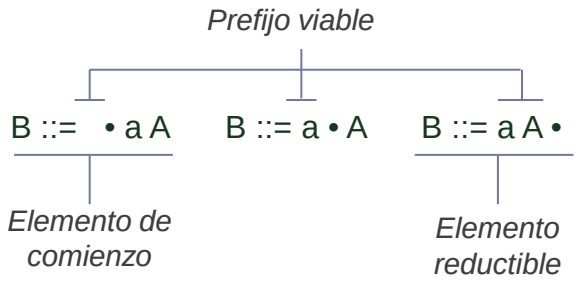
Elementos LR (0)

Un elemento es una regla de producción que tiene un punto (.) en algún lugar de la parte derecha de la regla. El punto se utiliza para identificar el estado de avance de procesamiento de la regla



Prefijo viable LR (0)

Un prefijo viable de un elemento corresponde con la forma de frase que queda a la izquierda del punto de dicho elemento. El prefijo viable identifica, de forma abstracta, la frase que se ha reconocido a la entrada y que valida, parcialmente, la selección de esa regla para proceder con la reducción



Analizadores sintácticos ascendentes

S' ::= S

S ::= (S) S |

Los diferentes tipos de analizadores sintácticos ascendentes – LR (0), SLR, LALR, LR (1) y LR (k) – sólo se diferencian entre sí por el procedimiento de construcción de las tablas acción e ir-A. En esta sección abordaremos su estudio

Conceptos esenciales LR (0) y SLR

Conjunto de elementos LR (0)

Un conjunto de elementos es una colección no ordenada de elementos. Los conjuntos de elementos se utilizan para caracterizar formalmente los estados potenciales por lo que atraviesa un compilador a lo largo del proceso de análisis

I = { S' ::= • S,
S ::= • (S) S,
S ::= • }

Colección canoníca de los conjuntos de elementos LR (0)

La colección de todos los conjuntos de elementos potenciales en los que se puede encontrar el analizador sintáctico se recoge en un conjunto llamado colección canónica de los conjuntos de elementos. Para su definición formal se utilizan dos funciones

- Cierre (I) — Se cierra un conjunto para incluir todos los elementos que pueden alcanzarse a partir de uno dado
- Ir- A (I, a) — Se genera un nuevo conjunto a partir de I y del simbolo terminal o no terminal a

Analizadores sintácticos ascendentes

Los diferentes tipos de analizadores sintácticos ascendentes – LR (0), SLR, LALR, LR (1) y LR (k) – sólo se diferencian entre sí por el procedimiento de construcción de las tablas acción e ir-A. En esta sección abordaremos su estudio

S' ::= S

S ::= (S) S |

Conceptos esenciales LR (0)

Función Cierre (I) LR (0)

El cierre de un conjunto I consiste en ampliar dicho conjunto con todos aquellos elementos con un punto al inicio de su parte derecha que son *accesibles* de forma directa o indirecta desde I

I_0 = Cierre ({ S' ::= • S }) = {

S' ::= • S,

S ::= • (S) S,

S ::= • }

Estar a punto de comenzar el procesamiento de S incluye estar a punto de procesar (S) S o

Cierre (I) = I

repetir

Para cada elemento A := α • B β ϵ Cierre (I) hacer

Para cada B ::= δ_1 | δ_2 | … | δ_n hacer

Cierre (I) = Cierre (I) $\cup$ {B := • δ_i} para todo i = 1..n

hasta no se pueden añadir más elementos a Cierre (I)

Analizadores sintácticos ascendentes

S' ::= S

S ::= (S) S |

Los diferentes tipos de analizadores sintácticos ascendentes – LR (0), SLR, LALR, LR (1) y LR (k) – sólo se diferencian entre sí por el procedimiento de construcción de las tablas acción e ir-A. En esta sección abordaremos su estudio

Conceptos esenciales LR (0)

Función Ir-A (I, a) LR (0)

Ir-A es una función que toma un estado y un elemento a Є N U T y devuelve un nuevo estado. La operación consiste en avanzar el punto una posición sobre todos aquellos elementos en los que a preceda al punto. En los que no ocurra así se eliminan del conjunto. Finalmente se cierra el conjunto resultante

Para todo elemento B ::= α • Aβ є I con A Є N U T hacer

Ir-A (I, A) = Cierre ({ B ::= α A • β })

Ir-A (I_0, '(') = Cierre ({ S ::= (• S) S })

= { S ::= (• S) S,

S ::= • (S) S,

S ::= • }

La aplicación de la función Ir-A (I_0, '(') indica cuál es la transición de estado que debe hacer el analizador sintáctico cuando se está en I_0 y llega un token '(' a la entrada

Analizadores sintácticos ascendentes

r_0. S' ::= S

r_1. S ::= B A end

r_2. A ::= begin C

r_3. C ::= codigo

r_4. B ::= tipo

r_5. B ::= id B

Los diferentes tipos de analizadores sintácticos ascendentes – LR (0), SLR, LALR, LR (1) y LR (k) – sólo se diferencian entre sí por el procedimiento de construcción de las tablas acción e ir-A. En esta sección abordaremos su estudio

Conceptos esenciales LR (0)

Colección canónica de los conjuntos de elementos C LR (0)

Se comienza por ampliar la gramática con la producción S' ::= S donde S es el axioma y S' es el nuevo axioma. Esto se hace para cubrir todas las alternativas de S. Después se aplica sistemáticamente el siguiente algoritmo hasta que C no pueda crecer más

Ampliar la gramática con S' ::= S

C = Cierre ({ S' ::= • S }) = { I_0 }

Para cada conjunto I_i ϵ C Hacer

  Para cada a ϵ T ∪ N Hacer

    Si existe un elemento B := α • a β ϵ I_i

      C = C ∪ Ir-A (I_i , a)

I0 = Cierre ({S' ::= •S}) = {S' ::= •S, S ::= •BA end, B::= •tipo, B ::= •id B}

I1 = IrA (I0, S) = Cierre ({S' ::= S•}) = {S' ::= S•}

I2 = IrA (I0, B) = Cierre ({S ::= B•A end}) = {S ::= B•A end, A ::= •begin C}

I3 = IrA (I0, tipo) = Cierre ({B ::= tipo•}) = {B ::= tipo•}

I4 = IrA (I0, id) = Cierre ({B ::= id•B}) = {B ::= id•B, B::= •tipo, B ::= •id B}

I5 = IrA (I2, A) = Cierre ({S ::= BA•end}) = {S ::= BA•end}

I6 = IrA (I2, begin) = Cierre ({A ::= begin•C}) = {A ::= begin•C, C ::= •codigo}

I7 = IrA (I4, B) = Cierre ({B ::= id B•}) = {B ::= id B•}

~~I8~~ = IrA (I4, tipo) = Cierre ({B ::= tipo•}) = {B ::= tipo•} = I3

~~I8~~ = IrA (I4, id) = Cierre ({B ::= id•B}) = {B ::= id•B, B::=•tipo, B::=•id B} = I4

I8 = IrA (I5, end) = Cierre ({S ::= BA end•}) = {S::= BA end•}

I9 = IrA (I6, C) = Cierre ({A ::= begin C•}) = {A ::= begin C•}

I10 = IrA (I6, codigo) = Cierre ({C ::= codigo•}) = {C ::= codigo•}

Analizadores sintácticos ascendentes

Los diferentes tipos de analizadores sintácticos ascendentes – LR (0), SLR, LALR, LR (1) y LR (k) – sólo se diferencian entre sí por el procedimiento de construcción de las tablas acción e ir-A. En esta sección abordaremos su estudio

Conceptos esenciales LR (0)

Autómata reconocedor de prefijos viables LR (0)

De manera similar se puede construir un autómata reconocedor de prefijos viables donde cada Ii se corresponde con un Si y cada transición etiquetada con a de S_i a S_j se corresponde con I_j = Ir-A (I_i, a)

S_0 = Cierre({ S' ::= • S })

Para cada S_i Hacer

  Para cada a ϵ N ∪ T Hacer

    Si Existe B ::= α • aβ ϵ S_i Entonces

      Crear estado nuevo S_n = Ir-A (S_i, A

      Crear transición de S_i a S_n etiquetada con a

r_0. S' ::= S

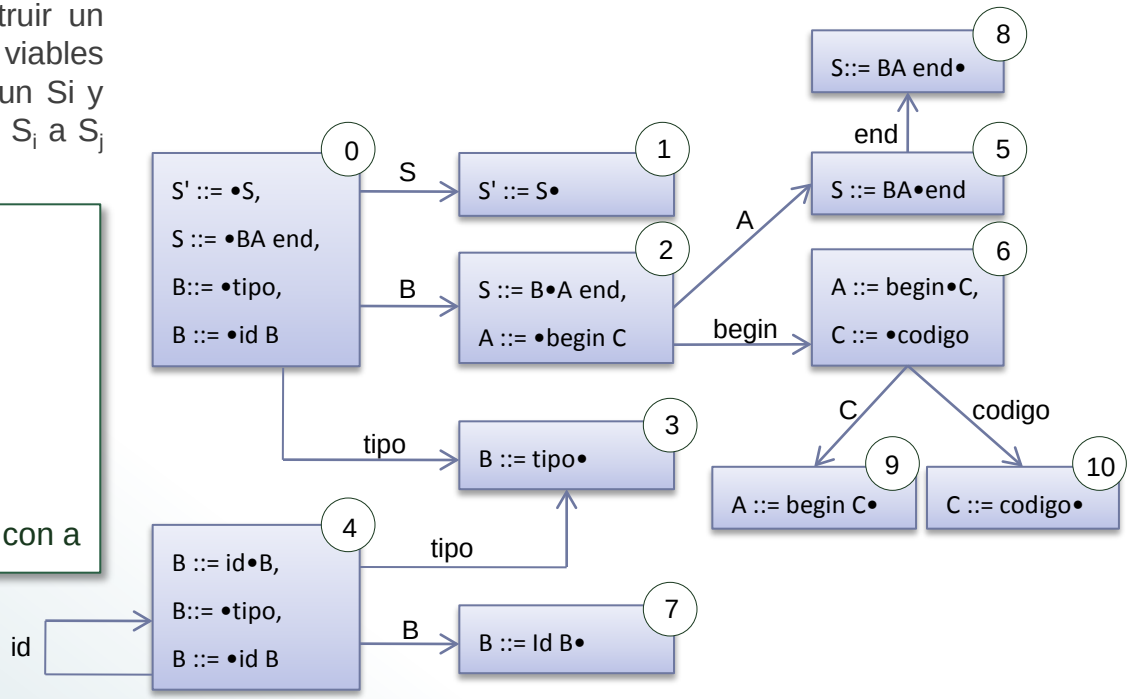
r_1. S ::= B A end

r_2. A ::= begin C

r_3. C ::= codigo

r_4. B ::= tipo

r_5. B ::= id B



Analizadores sintácticos ascendentes

Los diferentes tipos de analizadores sintácticos ascendentes – LR (0), SLR, LALR, LR (1) y LR (k) – sólo se diferencian entre sí por el procedimiento de construcción de las tablas acción e ir-A. En esta sección abordaremos su estudio

Construcción de las tablas LR (0)

Tablas LR (0) a partir de la colección canónica

A partir de la colección canónica de elementos es posible construir las tablas de los analizadores LR (0)

- Cada conjunto de elementos es un estado
- La tabla Ir-A (I, A) A Є N refleja la función Ir-A
- La tabla Ir-A (I, a) a Є T refleja los desplazamientos
- Para los elementos de reducción
  - Se rellena todas las celda con reducción

Obtener C = $\{I_0, I_1, \ldots, I_n\}$

Cada I_j є C se corresponde con el estado j del analizador

Construir la tabla Ir-A [s, A]

  Si Ir-A (I_i, A) = I_j, A Є N Entonces Ir-A [i, A] = j

Construir la tabla Acción [s, a]

  Para todo A ::= α • aβ Є I_i, a Є T Hacer

    Si Ir-A (I_i, a) = I_j Entonces Acción [i, a] = d_j

  Para todo A ::= β • o A ::= • Є I_i Hacer

    Acción [i, a] = r_k

  Si S' ::= S • Є I_i entonces Acción [i, $] = Aceptar

  Todas las demás entradas de Acción [k, s] = error

Analizadores sintácticos ascendentes

Los diferentes tipos de analizadores sintácticos ascendentes – LR (0), SLR, LALR, LR (1) y LR (k) – sólo se diferencian entre sí por el procedimiento de construcción de las tablas acción e ir-A. En esta sección abordaremos su estudio

Construcción de las tablas LR (0)

Tablas LR (0) a partir del autómata de prefijos viables

A partir del autómata reconocedor de prefijos viables es posible construir las tablas de los analizadores SLR

- Cada estado del autómata es un estado
- Transiciones S_i a S_j con A Є N reflejan Ir-A
- Transiciones S_i a S_j con a Є T reflejan los desplazamientos
- Para los elementos de reducción
  - Se calcula Siguientes del antecedente
  - Se rellena la cela con reducción por esa regla

Obtener El autómata reconocedor de prefijos viables

Cada estado S_j corresponde con el estado j

Construir la tabla Ir-A [s, A]

  Si existe transición de S_i a S_j con A, A Є N Entonces

    Ir-A [i, A] = j

Construir la tabla Acción [s, a]

  Para todo A ::= α • aβ Є S_i, a Є T Hacer

    Si existe transición de S_i a S_j con a entonces

      Acción [i, a] = d_j

  Para todo A ::= β • o A ::= • Є S_i Hacer

    Acción [i, a] = r_k

  Si S' ::= S • Є S_i entonces Acción [i, $] = Aceptar

  Todas las demás entradas de Acción [k, s] = error

Analizadores sintácticos ascendentes

Los diferentes tipos de analizadores sintácticos ascendentes – LR (0), SLR, LALR, LR (1) y LR (k) – sólo se diferencian entre sí por el procedimiento de construcción de las tablas acción e ir-A. En esta sección abordaremos su estudio

Construcción de las tablas LR (0)

Ejemplo

r_0. S' ::= S

r_1. S ::= B A end

r_2. A ::= begin C

r_3. C ::= codigo

r_4. B ::= tipo

r_5. B ::= id B

	end	begin	código	tipo	id	$	S	A	B	C
0							1		2	
1										
2								5		
3										
4									7	
5										
6										9
7										
8										
9										
10										

Se rellena la tabla Ir-A a partir de las transiciones del autómata etiquetadas con no terminales

Analizadores sintácticos ascendentes

Los diferentes tipos de analizadores sintácticos ascendentes – LR (0), SLR, LALR, LR (1) y LR (k) – sólo se diferencian entre sí por el procedimiento de construcción de las tablas acción e ir-A. En esta sección abordaremos su estudio

Construcción de las tablas LR (0)

Ejemplo

r_0. S' ::= S

r_1. S ::= B A end

r_2. A ::= begin C

r_3. C ::= codigo

r_4. B ::= tipo

r_5. B ::= id B

	end	begin	código	tipo	id	$	S	A	B	C
0				d3	d4		1		2	
1										
2		d6						5		
3										
4				d3	d4				7	
5	d8									
6			d10							9
7										
8										
9										
10										

Se rellenan los desplazamientos de la tabla de acción a partir de las transiciones del autómata etiquetadas con terminales

Analizadores sintácticos ascendentes

Los diferentes tipos de analizadores sintácticos ascendentes – LR (0), SLR, LALR, LR (1) y LR (k) – sólo se diferencian entre sí por el procedimiento de construcción de las tablas acción e ir-A. En esta sección abordaremos su estudio

Construcción de las tablas LR (0)

Ejemplo

r_0. S' ::= S

r_1. S ::= B A end

r_2. A ::= begin C

r_3. C ::= codigo

r_4. B ::= tipo

r_5. B ::= id B

	end	begin	código	tipo	id	$	S	A	B	C
0				d3	d4		1		2	
1						Ok				
2		d6						5		
3	r4	r4	r4	r4	r4	r4				
4				d3	d4				7	
5	d8									
6			d10							9
7	r5	r5	r5	r5	r5	r5				
8	r1	r1	r1	r1	r1	r1				
9	r2	r2	r2	r2	r2	r2				
10	r3	r3	r3	r3	r3	r3				

Se rellenan las reducciones y la aceptación de la tabla Acción a partir de los estados de reducción

Analizadores sintácticos ascendentes

Los diferentes tipos de analizadores sintácticos ascendentes – LR (0), SLR, LALR, LR (1) y LR (k) – sólo se diferencian entre sí por el procedimiento de construcción de las tablas acción e ir-A. En esta sección abordaremos su estudio

Construcción de las tablas LR (0)

Ejemplo

r_0. S' ::= S

r_1. S ::= B A end

r_2. A ::= begin C

r_3. C ::= codigo

r_4. B ::= tipo

r_5. B ::= id B

	end	begin	código	tipo	id	$	S	A	B	C
0	error	error	error	d3	d4	error	1		2	
1	error	error	error	error	error	Ok				
2	error	d6	error	error	error	error		5		
3	r4	r4	r4	r4	r4	r4				
4	error	error	error	d3	d4	error			7	
5	d8	error	error	error	error	error				
6	error	error	d10	error	error	error				9
7	r5	r5	r5	r5	r5	r5				
8	r1	r1	r1	r1	r1	r1				
9	r2	r2	r2	r2	r2	r2				
10	r3	r3	r3	r3	r3	r3				

El resto de casillas se rellenan con error

Analizadores sintácticos ascendentes

Los diferentes tipos de analizadores sintácticos ascendentes – LR (0), SLR, LALR, LR (1) y LR (k) – sólo se diferencian entre sí por el procedimiento de construcción de las tablas acción e ir-A. En esta sección abordaremos su estudio

Ventajas e inconvenientes de los analizadores LR (0)

Los analizadores sintácticos ascendentes del tipo LR (0) son los más sencillos de construir que existen pero el conjunto de gramáticas que reconocen es relativamente pequeño. En efecto, estos autómatas no utilizan ninguna estrategia predictiva en función de los terminales a la cabeza para dirigir los procesos de reducción

Ventajas	Inconvenientes
› Fácil de entender y construir	› Conjunto reducido de gramáticas
› La estrategia de reducción es sencilla	› No aplica estrategias predictivas
› Cada estado de reducción es por 1 regla	› No pueden existen 2 reducciones por estado
› Los errores se delegan a desplazamiento	› Mensajes de error menos localizados

Analizadores sintácticos ascendentes

Los diferentes tipos de analizadores sintácticos ascendentes – LR (0), SLR, LALR, LR (1) y LR (k) – sólo se diferencian entre sí por el procedimiento de construcción de las tablas acción e ir-A. En esta sección abordaremos su estudio

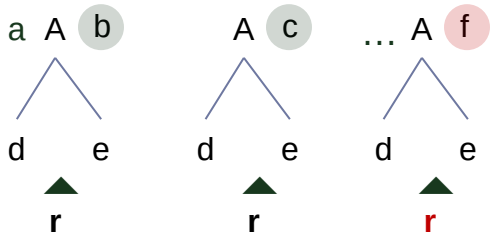
Desde el LR (0) hacia el SLR

Existe un tipo de analizadores sintácticos ascendentes más potentes que utiliza los mismos artefactos que el análisis LR (0) y que sin embargo es capaz de hacer reducciones predictivamente en función del terminal a la cabeza

S ::= a A b | A c

A ::= d e

Sólo si x Є Siguientes (A) aparece a la en la cabeza de lectura en el momento en curso es legítimo aplicar la reducción por el elemento reducible A ::= α •



Precisión de errores

Muchas celdas marcadas como reducción en LR (0) deberían ser consideradas como errores lo que redunda en una mayor precisión de los mensajes

Mayor precisión de análisis

Al predecir la reducción que debe producirse en función en la entrada se consigue afinar el proceso de análisis

Conjunto de gramáticas

Ahora en un mismo estado de puede haber celdas con distintas reducciones, con desplazamientos y con error con que las restricciones gramaticales no son tan fuertes

Analizadores sintácticos ascendentes

S' ::= S

S ::= (S) S |

Los diferentes tipos de analizadores sintácticos ascendentes – LR (0), SLR, LALR, LR (1) y LR (k) – sólo se diferencian entre sí por el procedimiento de construcción de las tablas acción e ir-A. En esta sección abordaremos su estudio

Conceptos esenciales SLR

Para refinar el proceso de análisis sintáctico, los analizadores ascendentes del tipo SLR hacen un uso extendido del autómata de prefijos viables y la colección canónica de elementos LR (0) Sin embargo, los conceptos que manejan ambos tipos de analizadores son esencialmente los mismos

Elemento SLR		**Elemento LR (0)**
Prefijo viable SLR		**Prefijo viable LR (0)**
Función cierre SLR	=	**Función cierre LR (0)**
Función Ir-A SLR	=	**Función Ir-A LR (0)**
Colección canónica SLR		**Colección canónica LR (0)**
Autómata SLR		**Autómata LR (0)**

Analizadores sintácticos ascendentes

Los diferentes tipos de analizadores sintácticos ascendentes – LR (0), SLR, LALR, LR (1) y LR (k) – sólo se diferencian entre sí por el procedimiento de construcción de las tablas acción e ir-A. En esta sección abordaremos su estudio

Construcción de las tablas SLR

Tablas SLR a partir de la colección canónica

A partir de la colección canónica de elementos es posible construir las tablas de los analizadores SLR

- Cada conjunto de elementos es un estado
- La tabla Ir-A (I, A) A Є N refleja la función Ir-A
- La tabla Ir-A (I, a) a Є T refleja los desplazamientos
- Para los elementos de reducción
  - Se calcula Siguientes del antecedente
  - Se rellena la celda con reducción por esa regla

Obtener C = $\{I_0, I_1, \ldots, I_n\}$

Cada I_j є C se corresponde con el estado j del analizador

Construir la tabla Ir-A [s, A]

Si Ir-A (I_i, A) = I_j, A Є N Entonces Ir-A [i, A] = j

Construir la tabla Acción [s, a]

Para todo A ::= α • aβ Є I_i, a Є T Hacer

Si Ir-A (I_i, a) = I_j Entonces Acción [i, a] = d_j

Para todo A ::= β • o A ::= • Є I_i Hacer

Para todo a Є Siguiente (A) hacer

Acción [i, a] = r_k

Si S' ::= S • Є I_i entonces Acción [i, $] = Aceptar

Todas las demás entradas de Acción [k, s] = error

Analizadores sintácticos ascendentes

Los diferentes tipos de analizadores sintácticos ascendentes – LR (0), SLR, LALR, LR (1) y LR (k) – sólo se diferencian entre sí por el procedimiento de construcción de las tablas acción e ir-A. En esta sección abordaremos su estudio

Construcción de las tablas SLR

Tablas SLR a partir del autómata de prefijos viables

A partir del autómata reconocedor de prefijos viables es posible construir las tablas de los analizadores SLR

- Cada estado del autómata es un estado
- Transiciones S_i a S_j con A Є N reflejan Ir-A
- Transiciones S_i a S_j con a Є T reflejan los desplazamientos
- Para los elementos de reducción
  - Se calcula Siguientes del antecedente
  - Se rellena la cela con reducción por esa regla

Obtener El autómata reconocedor de prefijos viables

Cada estado S_j corresponde con el estado j

Construir la tabla Ir-A [s, A]

  Si existe transición de S_i a S_j con A, A Є N Entonces

    Ir-A [i, A] = j

Construir la tabla Acción [s, a]

  Para todo A ::= α • aβ Є S_i, a Є T Hacer

    Si existe transición de S_i a S_j con a entonces

      Acción [i, a] = d_j

  Para todo A ::= β • o A ::= • Є S_i Hacer

    Para todo a Є Siguiente (A) hacer

      Acción [i, a] = r_k

  Si S' ::= S • Є S_i entonces Acción [i, $] = Aceptar

  Todas las demás entradas de Acción [k, s] = error

Analizadores sintácticos ascendentes

Los diferentes tipos de analizadores sintácticos ascendentes – LR (0), SLR, LALR, LR (1) y LR (k) – sólo se diferencian entre sí por el procedimiento de construcción de las tablas acción e ir-A. En esta sección abordaremos su estudio

Construcción de las tablas SLR

Ejemplo

r_0. S' ::= S

r_1. S ::= B A end

r_2. A ::= begin C

r_3. C ::= codigo

r_4. B ::= tipo

r_5. B ::= id B

	end	begin	código	tipo	id	$	S	A	B	C
0							1		2	
1										
2								5		
3										
4									7	
5										
6										9
7										
8										
9										
10										

Se rellena la tabla Ir-A a partir de las transiciones del autómata etiquetadas con no terminales

Analizadores sintácticos ascendentes

Los diferentes tipos de analizadores sintácticos ascendentes – LR (0), SLR, LALR, LR (1) y LR (k) – sólo se diferencian entre sí por el procedimiento de construcción de las tablas acción e ir-A. En esta sección abordaremos su estudio

Construcción de las tablas SLR

Ejemplo

r_0. S' ::= S

r_1. S ::= B A end

r_2. A ::= begin C

r_3. C ::= codigo

r_4. B ::= tipo

r_5. B ::= id B

	end	begin	código	tipo	id	$	S	A	B	C
0				d3	d4		1		2	
1										
2		d6						5		
3										
4				d3	d4				7	
5	d8									
6			d10							9
7										
8										
9										
10										

Se rellenan los desplazamientos de la tabla de acción a partir de las transiciones del autómata etiquetadas con terminales

Analizadores sintácticos ascendentes

Los diferentes tipos de analizadores sintácticos ascendentes – LR (0), SLR, LALR, LR (1) y LR (k) – sólo se diferencian entre sí por el procedimiento de construcción de las tablas acción e ir-A. En esta sección abordaremos su estudio

r_0. S' ::= S

r_1. S ::= B A end

r_2. A ::= begin C

r_3. C ::= codigo

r_4. B ::= tipo

r_5. B ::= id B

Construcción de las tablas SLR

Ejemplo

	end	begin	código	tipo	id	$	S	A	B	C
0				d3	d4		1		2	
1						Ok				
2		d6						5		
3		r4								
4				d3	d4				7	
5	d8									
6			d10							9
7		r5								
8						r1				
9	r2									
10	r3									

Siguiente (B) = Primero (A end)
= Primero (A) = { begin }

Se rellenan las reducciones y la aceptación de la tabla Acción a partir de los estados de reducción y la información de los conjuntos Siguientes

Siguiente (C) = Siguiente (A) = { end }

Siguiente (S) = Siguiente (S') = { $ }

Analizadores sintácticos ascendentes

Los diferentes tipos de analizadores sintácticos ascendentes – LR (0), SLR, LALR, LR (1) y LR (k) – sólo se diferencian entre sí por el procedimiento de construcción de las tablas acción e ir-A. En esta sección abordaremos su estudio

Construcción de las tablas SLR

Ejemplo

r_0. S' ::= S

r_1. S ::= B A end

r_2. A ::= begin C

r_3. C ::= codigo

r_4. B ::= tipo

r_5. B ::= id B

	end	begin	código	tipo	id	$	S	A	B	C
0	error	error	error	d3	d4	error	1		2	
1	error	error	error	error	error	Ok				
2	error	d6	error	error	error	error		5		
3	error	r4	error	error	error	error				
4	error	error	error	d3	d4	error			7	
5	d8	error	error	error	error	error				
6	error	error	d10	error	error	error				9
7	error	r5	error	error	error	error				
8	error	error	error	error	error	r1				
9	r2	error	error	error	error	error				
10	r3	error	error	error	error	error				

El resto de casillas se rellenan con error

Analizadores sintácticos ascendentes

Los diferentes tipos de analizadores sintácticos ascendentes – LR (0), SLR, LALR, LR (1) y LR (k) – sólo se diferencian entre sí por el procedimiento de construcción de las tablas acción e ir-A. En esta sección abordaremos su estudio

Ventajas e inconvenientes de los analizadores SLR

Los analizadores sintácticos ascendentes del tipo SLR suponen una mejora con respecto a los del tipo LR (0) que refinan las condiciones de entrada que deben darse para poder hacer una reducción por una determinada regla

Ventajas	Inconvenientes
› Fácil de entender y construir	› Conjunto mayor pero reducido de gramáticas
› Predice reducción mediante Siguientes	› Estrategia de reducción más compleja
› Varias reducciones por el mismo estado	› No existen estados de (única) reducción
› Utiliza el mismo autómata LR (0)	› El autómata LR (0) no contempla predicción

Analizadores sintácticos ascendentes

Los diferentes tipos de analizadores sintácticos ascendentes – LR (0), SLR, LALR, LR (1) y LR (k) – sólo se diferencian entre sí por el procedimiento de construcción de las tablas acción e ir-A. En esta sección abordaremos su estudio

Desde el SLR hacia el LR (1)

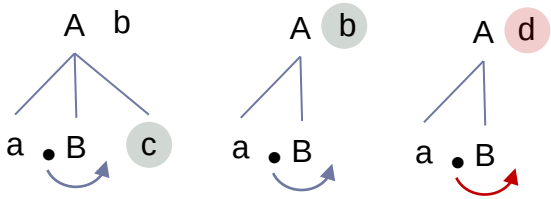
El análisis predictivo que realizan los analizadores ascendentes del tipo SLR resulta demasiado tosco ya que aún se pueden afinar más los criterios de aplicación predictiva de desplazamientos

S ::= A b | c B d

A ::= a B c | a B

B ::= d e

No todos los x Є T, ni siquiera los x Є Siguientes (B) a la cabeza de la entrada en un momento dado son candidatos viables para hacer un avance por el elemento A ::= α • B γ



Conjunto de gramáticas

Además de las características destacadas anteriormente, ahora la caracterización del conjunto de terminales que deben aparecer a la entrada para hacer un paso de derivación se ve más afinado. Esto impone menos restricciones gramaticales lo que permite articular un gran número de gramáticas

Precisión de errores

Como consecuencia también ocurre que la identificación de los tipos de errores, y sus mensajes asociados mejora ya que cada uno describe más precisamente el contexto sintáctico en el que se produce y los terminales que se deberían haber esperado

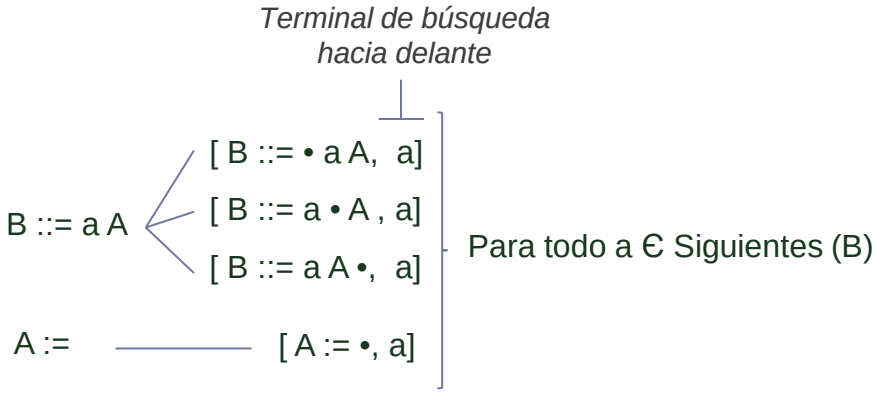
Analizadores sintácticos ascendentes

Los diferentes tipos de analizadores sintácticos ascendentes – LR (0), SLR, LALR, LR (1) y LR (k) – sólo se diferencian entre sí por el procedimiento de construcción de las tablas acción e ir-A. En esta sección abordaremos su estudio

Conceptos esenciales LR (1)

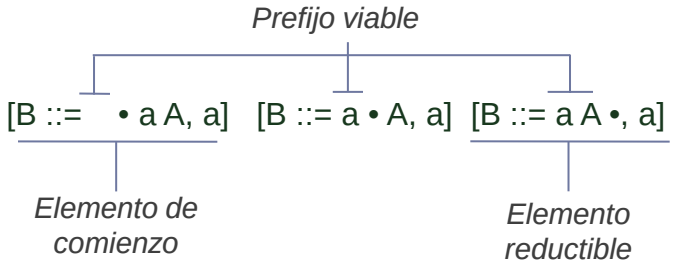
Elementos LR (1)

Un elemento LR (1) es un par formado por un elemento LR (0) y un terminal de búsqueda hacia delante. La idea es acarrear en cada elemento el terminal que debe encontrarse a la entrada a la hora de realizar la reducción. Debe pertenecer al conjunto Siguiente del antecedente



Prefijo viable LR (1)

Un prefijo viable de un elemento corresponde con la forma de frase que queda a la izquierda del punto de dicho elemento. El prefijo viable identifica, de forma abstracta, la frase que se ha reconocido a la entrada y que valida, parcialmente, la selección de esa regla para proceder con la reducción



Analizadores sintácticos ascendentes

Los diferentes tipos de analizadores sintácticos ascendentes – LR (0), SLR, LALR, LR (1) y LR (k) – sólo se diferencian entre sí por el procedimiento de construcción de las tablas acción e ir-A. En esta sección abordaremos su estudio

S' ::= S

S ::= (S) | a

Conceptos esenciales LR (1)

Conjunto de elementos LR (1)

Un conjunto de elementos es una colección no ordenada de elementos. Los conjuntos de elementos se utilizan para caracterizar formalmente los estados potenciales por lo que atraviesa un compilador a lo largo del proceso de análisis

I = { [S' ::= • S , $],
[S ::= • (S) , $],
[S ::= • a , $] }

Colección canoníca de los conjuntos de elementos LR (1)

La colección de todos los conjuntos de elementos potenciales en los que se puede encontrar el analizador sintáctico se recoge en un conjunto llamado colección canónica de los conjuntos de elementos. Para su definición formal se utilizan dos funciones

- Cierre (I) — Se cierra un conjunto para incluir todos los elementos que pueden alcanzarse a partir de uno dado
- Ir- A (I, a) — Se genera un nuevo conjunto a partir de I y del simbolo terminal o no terminal a

Analizadores sintácticos ascendentes

Los diferentes tipos de analizadores sintácticos ascendentes – LR (0), SLR, LALR, LR (1) y LR (k) – sólo se diferencian entre sí por el procedimiento de construcción de las tablas acción e ir-A. En esta sección abordaremos su estudio

S' ::= S

S ::= (S) | a

Conceptos esenciales LR (1)

Función Cierre (I) LR (1)

El cierre de un conjunto I consiste en ampliar dicho conjunto con todos aquellos elementos con un punto al inicio de su parte derecha que son *accesibles* de forma directa o indirecta desde I. Además debe actualizarse el terminal de búsqueda hacia delante

Cierre (I) = I

repetir

  Para cada elemento [A := α • B β, a] ϵ Cierre (I) hacer

    Para cada B ::= δ_1 | δ_2 | … | δ_n hacer

      Para cada b ϵ Primeros (β a)

        Cierre (I) = Cierre (I) $\cup$ { [B := • δ_i, b] } para todo i = 1..n

hasta no se pueden añadir más elementos a Cierre (I)

I_0 = Cierre ({ [S' ::= • S, $] }) =

{ [S' ::= • S, $],

[S ::= • (S) , $],

[S ::= • a , $] }

I_2 = Cierre ({ [S' ::= (• S), $] }) =

{ [S' ::= (• S), $],

[S ::= • (S), ')'],

[S ::= • a, ')'] }

Analizadores sintácticos ascendentes

Los diferentes tipos de analizadores sintácticos ascendentes – LR (0), SLR, LALR, LR (1) y LR (k) – sólo se diferencian entre sí por el procedimiento de construcción de las tablas acción e ir-A. En esta sección abordaremos su estudio

S' ::= S

S ::= (S) | a

Conceptos esenciales LR (1)

Función Ir-A (I, a) LR (1)

Ir-A es una función que toma un estado y un elemento a Є N U T y devuelve un nuevo estado. La operación consiste en avanzar el punto una posición sobre todos aquellos elementos en los que a preceda al punto. En los que no ocurra así se eliminan del conjunto. Finalmente se cierra el conjunto resultante

Para todos los elementos B := [α • Aβ, b] є I hacer

Ir-A (I, A) = Cierre ({ B := α A • β })

Ir-A (I_0, '(') = Cierre ({ [S ::= (• S), $] })

= { [S ::= (• S), $],

[S ::= • (S),)],

[S ::= • a,)] }

Analizadores sintácticos ascendentes

r_1. S' ::= S

r_2. S ::= C C

r_3. C ::= c C

r_4. C::= d

Los diferentes tipos de analizadores sintácticos ascendentes – LR (0), SLR, LALR, LR (1) y LR (k) – sólo se diferencian entre sí por el procedimiento de construcción de las tablas acción e ir-A. En esta sección abordaremos su estudio

Conceptos esenciales LR (1)

Colección canónica de los conjuntos de elementos C LR (1)

Continúa…

Se comienza por ampliar la gramática con la producción S' ::= S donde S es el axioma y S' es el nuevo axioma. Esto se hace para cubrir todas las alternativas de S. Después se aplica sistemáticamente el siguiente algoritmo hasta que C no pueda crecer más

Ampliar la gramática con S' ::= S

C = Cierre ({ S' ::= • S }) = { I_0 }

Para cada conjunto I_i ϵ C Hacer

  Para cada a ϵ T ∪ N Hacer

    Si existe un elemento B := α • a β ϵ I_i

      C = C ∪ Ir-A (I_i , a)

I0 = Cierre ({ [S' ::= •S, $] })

  = {[S' ::= •S, $], [S ::= • CC, $],

    [C ::= • cC, c],

    [C ::= • cC, d],

    [C ::= •d, c],

    [C ::= •d, d] }

I1 = Ir-A (I0, S)

  = Cierre ({ [S' ::= S •, $] })

  = { [S' ::= S •, $] }

I2 = Ir-A (I0, C)

  = Cierre ({ [S ::= C •C,$] })

  = { [S ::= C•C,$],

    [C::= •cC, $],

    [C::= •d, $] }

I3 = Ir-A (I0, c)

  = Cierre ({ [C ::= c•C, c] [C ::= c•C, d] })

  = { [C ::= c•C, c],

    [C::= •cC, c],

    [C ::= •d, c],

    [C ::= c•C, d],

    [C::= •cC, d],

    [C ::= •d, d] }

I4 = Ir-A (I0, d)

  = Cierre ({ [C ::= d •, c] [C::= d •, d] })

  = { [C ::= d •, c] [C::= d •, d] }

I5 = Ir-A (I2,C)

  = Cierre ({ [S ::= CC•, $] })

  = { [S ::= CC•, $] }

Analizadores sintácticos ascendentes

r_1. S' ::= S

r_2. S ::= C C

r_3. C ::= c C

r_4. C::= d

Los diferentes tipos de analizadores sintácticos ascendentes – LR (0), SLR, LALR, LR (1) y LR (k) – sólo se diferencian entre sí por el procedimiento de construcción de las tablas acción e ir-A. En esta sección abordaremos su estudio

Conceptos esenciales LR (1)

Colección canónica de los conjuntos de elementos C LR (1)

Se comienza por ampliar la gramática con la producción S' ::= S donde S es el axioma y S' es el nuevo axioma. Esto se hace para cubrir todas las alternativas de S. Después se aplica sistemáticamente el siguiente algoritmo hasta que C no pueda crecer más

Ampliar la gramática con S' ::= S

C = Cierre ({ S' ::= • S }) = { I_0 }

Para cada conjunto I_i ϵ C Hacer

  Para cada a ϵ T ∪ N Hacer

    Si existe un elemento B := α • a β ϵ I_i

      C = C ∪ Ir-A (I_i , a)

I6 = Ir-A (I2, c)

  = Cierre ({ [C::=c •C,$] })

  = { [C ::= c •C,$],

    [C: := •cC, $],

    [C ::= •d, $] }

I7 = Ir-A (I2, d)

  = Cierre ({ [C::= d •, $] })

  = { [C ::= d •,$] }

I8 = Ir-A (I3, C)

  = Cierre ({ [C ::= cC •, c],

    [C ::= cC •, d] })

  = { [C ::= cC •,c],

    [C ::= cC •,d] }

~~I9~~ = Ir-A (I3, c)

  = Cierre({ [C::= c•C, c], [C ::= c•C, d] }) = I3

~~I9~~ = Ir-A (I3, d)

  = Cierre ({ [C ::= d •, c] [C::= d •, d] }) = I4

I9 = Ir-A (I6, C)

  = Cierre({ [C ::= cC•, $] }) = { [C ::= cC•, $] }

~~I10~~ = Ir-A (I6, c)

  = Cierre ({ [C::=c•C, $] }) = I6

~~I10~~ = Ir-A (I6, d)

  = Cierre ({ [C::= d•, $] }) = I7

Analizadores sintácticos ascendentes

Los diferentes tipos de analizadores sintácticos ascendentes – LR (0), SLR, LALR, LR (1) y LR (k) – sólo se diferencian entre sí por el procedimiento de construcción de las tablas acción e ir-A. En esta sección abordaremos su estudio

r_1. S' ::= S

r_2. S ::= C C

r_3. C ::= c C

r_4. C::= d

Conceptos esenciales LR (1)

Autómata reconocedor de prefijos viables LR (1)

De manera similar se puede construir un autómata reconocedor de prefijos viables donde cada Ii se corresponde con un Si y cada transición etiquetada con a de S_i a S_j se corresponde con I_j = Ir-A (I_i, a)

S_0 = Cierre({ S' ::= • S })

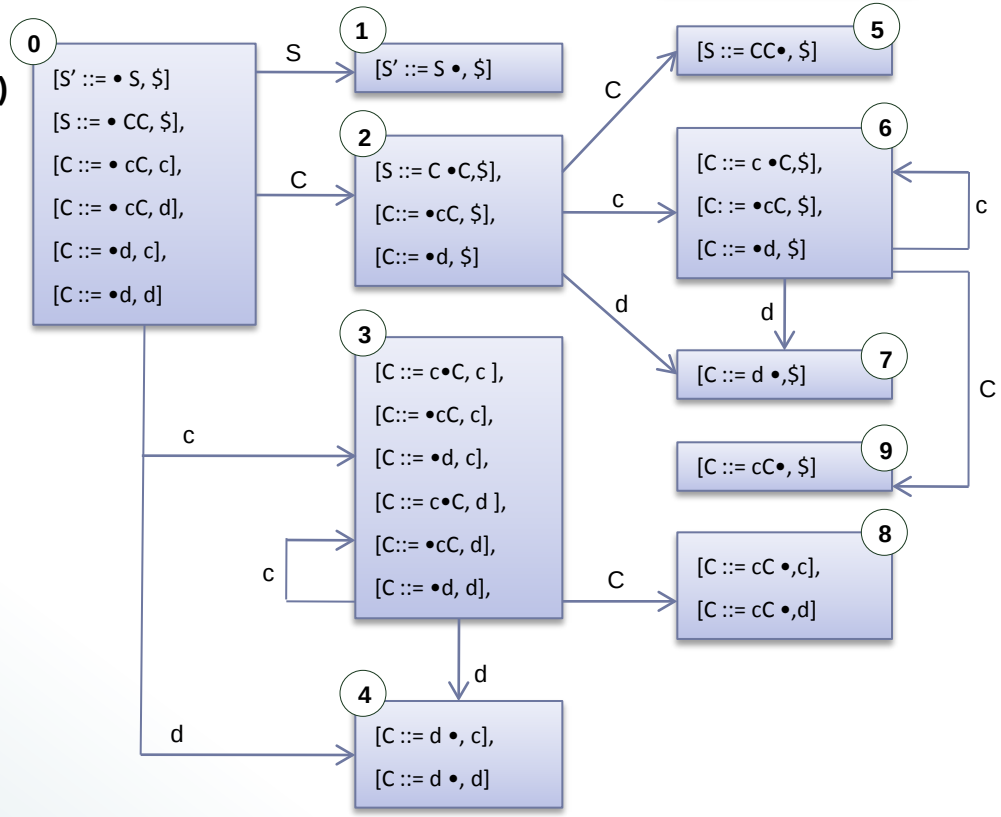
Para cada S_i Hacer

  Para cada a ϵ N ∪ T Hacer

    Si Existe B ::= α • aβ ϵ S_i Entonces

      Crear estado nuevo S_n = Ir-A (S_i, A

      Crear transición de S_i a S_n etiquetada con a



Analizadores sintácticos ascendentes

Los diferentes tipos de analizadores sintácticos ascendentes – LR (0), SLR, LALR, LR (1) y LR (k) – sólo se diferencian entre sí por el procedimiento de construcción de las tablas acción e ir-A. En esta sección abordaremos su estudio

Construcción de las tablas LR (1)

Tablas LR (1) a partir de la colección canónica

A partir de la colección canónica de elementos es posible construir las tablas de los analizadores LR (1)

- Cada conjunto de elementos es un estado
- La tabla Ir-A (I, A) A Є N refleja la función Ir-A
- La tabla Ir-A (I, a) a Є T refleja los desplazamientos
- Para los elementos de reducción
  - Se mira el terminal de búsqueda adelantada
  - Se rellena la celda con reducción por esa regla

Obtener C = $\{I_0, I_1, \ldots, I_n\}$

Cada I_j є C se corresponde con el estado j del analizador

Construir la tabla Ir-A [s, A]

Si Ir-A (I_i, A) = I_j, A Є N Entonces Ir-A [i, A] = j

Construir la tabla Acción [s, a]

Para todo [A ::= α • aβ, b] Є I_i, a Є T Hacer

Si Ir-A (I_i, a) = I_j Entonces Acción [i, a] = d_j

Para todo [A ::= α •, a] Є I_i a Є T Hacer

Acción [i, a] = r_k

Si [S' ::= S •, $] Є I_i entonces Acción [i, $] = Aceptar

Todas las demás entradas de Acción [k, s] = error

Analizadores sintácticos ascendentes

Los diferentes tipos de analizadores sintácticos ascendentes – LR (0), SLR, LALR, LR (1) y LR (k) – sólo se diferencian entre sí por el procedimiento de construcción de las tablas acción e ir-A. En esta sección abordaremos su estudio

Construcción de las tablas LR (1)

Tablas LR (1) a partir del autómata de prefijos viables

A partir del autómata reconocedor de prefijos viables es posible construir las tablas de los analizadores SLR

- Cada estado del autómata es un estado
- Transiciones S_i a S_j con A Є N reflejan Ir-A
- Transiciones S_i a S_j con a Є T reflejan los desplazamientos
- Para los elementos de reducción
  - Se mira el terminal de búsqueda adelantada
  - Se rellena la cela con reducción por esa regla

Obtener El autómata reconocedor de prefijos viables

Cada estado S_j corresponde con el estado j

Construir la tabla Ir-A [s, A]

  Si existe transición de S_i a S_j con A, A Є N Entonces

    Ir-A [i, A] = j

Construir la tabla Acción [s, a]

  Para todo [A ::= α • aβ, b] Є S_i, a Є T Hacer

    Si existe transición de S_i a S_j con a entonces

      Acción [i, a] = d_j

  Para todo [A ::= α •, a] Є S_i a Є T Hacer

    Acción [i, a] = r_k

  Si [S' ::= S •, $] Є S_i Acción [i, $] = Aceptar

  Todas las demás entradas de Acción [k, s] = error

Analizadores sintácticos ascendentes

r_1. S' ::= S

r_2. S ::= C C

r_3. C ::= c C

r_4. C::= d

Los diferentes tipos de analizadores sintácticos ascendentes – LR (0), SLR, LALR, LR (1) y LR (k) – sólo se diferencian entre sí por el procedimiento de construcción de las tablas acción e ir-A. En esta sección abordaremos su estudio

Construcción de las tablas LR (1)

Ejemplo

	c	d	$	S	C
0				1	2
1					
2					5
3					8
4					
5					
6					9
7					
8					
9					

	c	d	$	S	C
0	d3	d4		1	2
1					
2	d6	d7			5
3	d3	d4			8
4					
5					
6	d6	d7			9
7					
8					
9					

Analizadores sintácticos ascendentes

r_1. S' ::= S

r_2. S ::= C C

r_3. C ::= c C

r_4. C::= d

Los diferentes tipos de analizadores sintácticos ascendentes – LR (0), SLR, LALR, LR (1) y LR (k) – sólo se diferencian entre sí por el procedimiento de construcción de las tablas acción e ir-A. En esta sección abordaremos su estudio

Construcción de las tablas LR (1)

Ejemplo

	c	d	$	S	C
0	d3	d4		1	2
1					
2	d6	d7			5
3	d3	d4			8
4	r4	r4			
5	r2	r2			
6	d6	d7			9
7			r4		
8	r3	r3			
9			r3		

	c	d	$	S	C
0	d3	d4	error	1	2
1	error	error	ok		
2	d6	d7	error		5
3	d3	d4	error		8
4	r4	r4	error		
5	r2	r2	error		
6	d6	d7	error		9
7	error	error	r4		
8	r3	r3	error		
9	error	error	r3		

Analizadores sintácticos ascendentes

Los diferentes tipos de analizadores sintácticos ascendentes – LR (0), SLR, LALR, LR (1) y LR (k) – sólo se diferencian entre sí por el procedimiento de construcción de las tablas acción e ir-A. En esta sección abordaremos su estudio

Ventajas e inconvenientes de los analizadores LR (1)

Los analizadores sintácticos ascendentes del tipo LR (1) son los más generales que existen y admiten un gran número de gramáticas. No obstante la complejidad de los autómatas reconocedores de prefijos viables os hacen inaplicables en la práctica

Ventajas	Inconvenientes
› Gran conjunto de gramáticas posibles	› Mas difícil de entender y construir
› Predicción con terminales de búsqueda	› Estrategia de reducción más compleja
› Varias reducciones por el mismo estado	› No existen estados de (únicos) de reducción
› La predicción está muy ajustada	› Utiliza un autómata muy complejo

Analizadores sintácticos ascendentes

Los diferentes tipos de analizadores sintácticos ascendentes – LR (0), SLR, LALR, LR (1) y LR (k) – sólo se diferencian entre sí por el procedimiento de construcción de las tablas acción e ir-A. En esta sección abordaremos su estudio

Desde el LR (1) hacia el LALR

El único inconveniente del análisis LR (1) es la explosión combinatoria de estados que surge en el autómata reconocedor de prefijos viables cuando se tienen en cuenta los terminales de búsqueda adelantada. LALR simplifica estos autómatas a partir de 2 principios

Primer principio del análisis LALR

El núcleo de un estado de un autómata reconocedor de prefijos viables LR (1) es igual al estado equivalente de LR (0)

Segundo principio del análisis LALR

Dados 2 estados s_1 y s_2 de un autómata LR (1) con el mismo núcleo. Si s_1 tiene una transición con X a t_1, entonces s_2 también lo tiene con X a algún t_2 y t_1 y t_2 comparten el mismo núcleo

I = { [S' ::= • S , $],
[S ::= • (S) , $],
[S ::= • a , $] }

Llamamos núcleo de un estado al conjunto formado por los primeros componentes de cada elemento de dicho estado

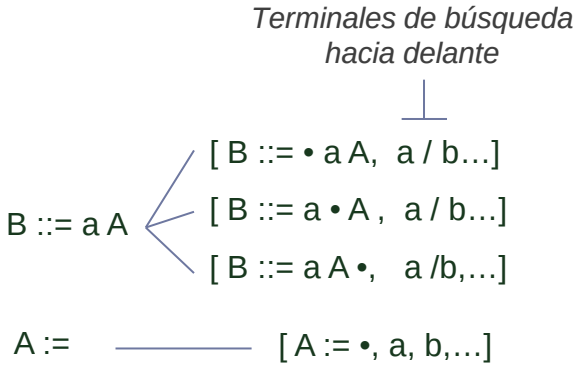
Analizadores sintácticos ascendentes

Los diferentes tipos de analizadores sintácticos ascendentes – LR (0), SLR, LALR, LR (1) y LR (k) – sólo se diferencian entre sí por el procedimiento de construcción de las tablas acción e ir-A. En esta sección abordaremos su estudio

Conceptos esenciales LALR

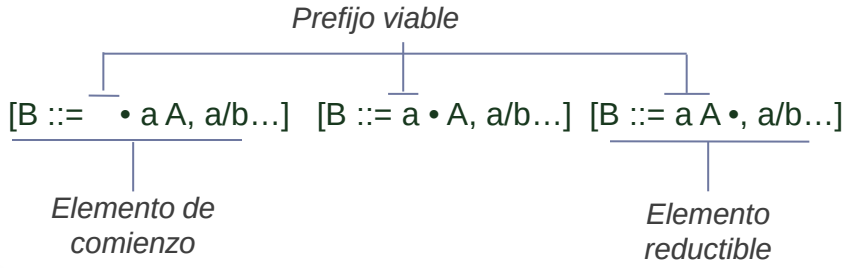
Elementos LALR

Un elemento LR (1) es un par formado por un elemento LR (0) y un o varios terminales de búsqueda hacia delante. La idea es acarrear en cada elemento el/los terminales que deben encontrarse a la entrada a la hora de realizar la reducción. Debe pertenecer al conjunto Siguiente del antecedente



Prefijo viable LALR

Un prefijo viable de un elemento corresponde con la forma de frase que queda a la izquierda del punto de dicho elemento. El prefijo viable identifica, de forma abstracta, la frase que se ha reconocido a la entrada y que valida, parcialmente, la selección de esa regla para proceder con la reducción



Analizadores sintácticos ascendentes

Los diferentes tipos de analizadores sintácticos ascendentes – LR (0), SLR, LALR, LR (1) y LR (k) – sólo se diferencian entre sí por el procedimiento de construcción de las tablas acción e ir-A. En esta sección abordaremos su estudio

S' ::= S

S ::= (S) | a

Conceptos esenciales LALR

Conjunto de elementos LALR

Un conjunto de elementos es una colección no ordenada de elementos. Los conjuntos de elementos se utilizan para caracterizar formalmente los estados potenciales por lo que atraviesa un compilador a lo largo del proceso de análisis

I = { [S' ::= • S , $],
[S ::= • (S) , $],
[S ::= • a , $] }

Función Cierre (I) LALR

El cierre de un conjunto I consiste en ampliar dicho conjunto con todos aquellos elementos con un punto al inicio de su parte derecha que son *accesibles* de forma directa o indirecta desde I. Además debe actualizarse el terminal de búsqueda hacia delante

Cierre (I) LALR = **Cierre (I) LR (1)**

Función Ir-A (I, a) LALR

Ir-A es una función que toma un estado y un elemento a Є N U T y devuelve un nuevo estado. La operación consiste en avanzar el punto una posición sobre todos aquellos elementos en los que a preceda al punto. En los que no ocurra así se eliminan del conjunto. Finalmente se cierra el conjunto resultante

Ir-A (I, a) LALR = **Ir-A (I, a) LR (1)**

Analizadores sintácticos ascendentes

r_1. S' ::= S

r_2. S ::= C C

r_3. C ::= c C

r_4. C::= d

Los diferentes tipos de analizadores sintácticos ascendentes – LR (0), SLR, LALR, LR (1) y LR (k) – sólo se diferencian entre sí por el procedimiento de construcción de las tablas acción e ir-A. En esta sección abordaremos su estudio

Conceptos esenciales LALR

Colección canónica de los conjuntos de elementos C LALR

Todos los conjuntos de elementos con el mismo núcleo se fusionan en un mismo conjunto de elementos. Se devuelve la colección canónica resultante

```
Ampliar la gramática con S' ::= S
C = Cierre ( { S' ::= • S } ) = { I0 }
C' = { }
Mientras (C != { } )
    I = c.extraer ()
    Js = c.buscarMismoNucleo (I)
    Para todo J en Js
        I = I U J
    C = C U { I }
```

```
I0 =  { [S ::= • CC, $],
        [C ::= • cC, c/d],
        [C ::= •d, c/d]  }
I1 =  { [S' ::= S •, $] }
I2 =  { [S ::= C •C,$],
        [C::= •cC, $],
        [C::= •d, $]  }
I3 = { [C ::= c•C, c/d/$ ],
        [C::= •cC, c/d/$],
        [C ::= •d, c/d/$]  } = I6
```

```
I4 = { [C ::= d •, c/d/$]  }  = I7
I5 = { [C ::= CC•, $] }
I8 = { [C ::= cC•, $] } = I9
```

Analizadores sintácticos ascendentes

r_1. S' ::= S

r_2. S ::= C C

r_3. C ::= c C

r_4. C::= d

Los diferentes tipos de analizadores sintácticos ascendentes – LR (0), SLR, LALR, LR (1) y LR (k) – sólo se diferencian entre sí por el procedimiento de construcción de las tablas acción e ir-A. En esta sección abordaremos su estudio

Conceptos esenciales LALR

Autómata reconocedor de prefijos viables LALR

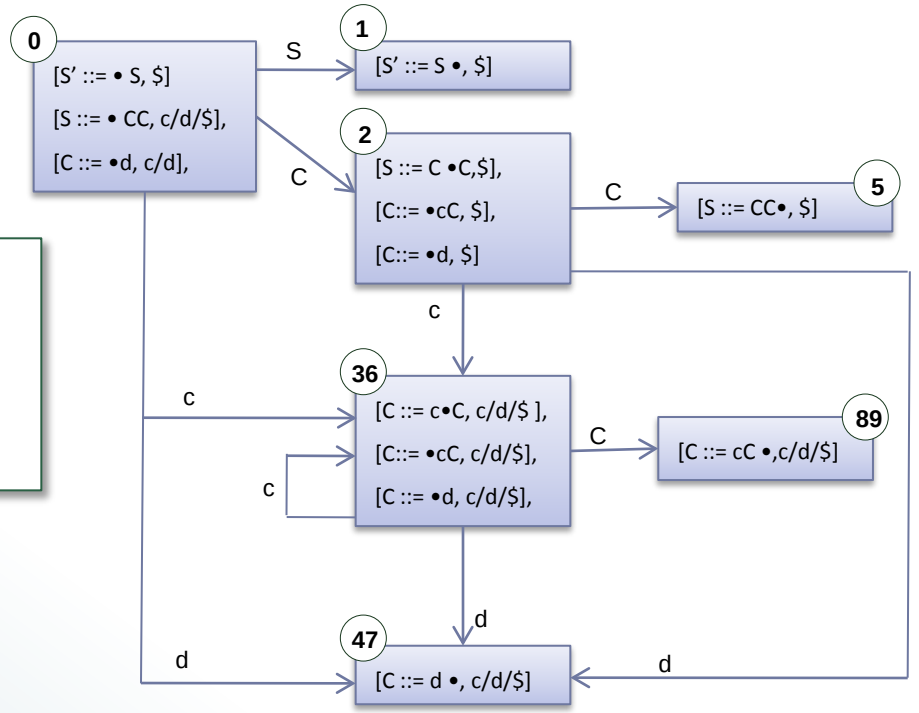
Todos los conjuntos de elementos con el mismo núcleo se fusionan en un mismo conjunto de elementos. Se devuelve la colección canónica resultante

Hacer el autómata para LR (1)

Para cada subconjunto S_1 … S_k con el mismo núcleo

$S'_1 = U\ S_i$

Fusionar también todas las transiciones



Analizadores sintácticos ascendentes

Los diferentes tipos de analizadores sintácticos ascendentes – LR (0), SLR, LALR, LR (1) y LR (k) – sólo se diferencian entre sí por el procedimiento de construcción de las tablas acción e ir-A. En esta sección abordaremos su estudio

Construcción de las tablas LALR

Tablas LALR a partir de la colección canónica

A partir de la colección canónica de elementos es posible construir las tablas de los analizadores LALR

- Cada conjunto de elementos es un estado
- La tabla Ir-A (I, A) A Є N refleja la función Ir-A
- La tabla Ir-A (I, a) a Є T refleja los desplazamientos
- Para los elementos de reducción
  - Se mira el terminal de búsqueda adelantada
  - Se rellena la celda con reducción por esa regla
- Se fusionan los resultados de la tabla acción

Obtener C = $\{I_0, I_1, \ldots, I_n\}$ de LR (1)

Obtener C' = $\{J_0, J_1, \ldots, J_k\}$ de LALR

Construir la tabla Ir-A [s, A]

Si $J_i = \{ I_0,, \ldots, I_m \}$ Entonces

Ir-A (J_i, a) = U Ir-A (I_i, a)

Construir la tabla Acción [s, a]

Construir la tabla Acción [s, a] de LR (1)

Fusionar todas las celdas de las filas con estados fusionados

Si aparecen conflictos G no es LALR

Analizadores sintácticos ascendentes

Los diferentes tipos de analizadores sintácticos ascendentes – LR (0), SLR, LALR, LR (1) y LR (k) – sólo se diferencian entre sí por el procedimiento de construcción de las tablas acción e ir-A. En esta sección abordaremos su estudio

Construcción de las tablas LALR

Tablas LALR a partir del autómata de prefijos viables

A partir del autómata reconocedor de prefijos viables es posible construir las tablas de los analizadores LALR

- Cada estado del autómata es un estado
- Transiciones S_i a S_j con A Є N reflejan Ir-A
- Transiciones S_i a S_j con a Є T reflejan los desplazamientos
- Para los elementos de reducción
  - Se mira el terminal de búsqueda adelantada
  - Se rellena la cela con reducción por esa regla
- Se fusionan los resultados de la tabla acción

Obtener el autómata LR (1) con estados T_j

Obtener el autómata LALR con estados S_i

Construir la tabla Ir-A [s, A]

Si $S_i = \{ T_a, \ldots, T_b \}$, $S_j = \{ T_c, \ldots, T_d \}$ Entonces

Si existe alguna transición de Tv Є S_i a T_r Є S_j con X

Trazar transición de S_i a S_{ij} con X

Construir la tabla Acción [s, a]

Construir la tabla Acción [s, a] de LR (1)

Fusionar toda celda de filas con estados fusionados

Si aparecen conflictos G no es LALR

Analizadores sintácticos ascendentes

r_1. S' ::= S

r_2. S ::= C C

r_3. C ::= c C

r_4. C::= d

Los diferentes tipos de analizadores sintácticos ascendentes – LR (0), SLR, LALR, LR (1) y LR (k) – sólo se diferencian entre sí por el procedimiento de construcción de las tablas acción e ir-A. En esta sección abordaremos su estudio

Construcción de las tablas LALR

Ejemplo

	c	d	$	S	C
0				1	2
1					
2					5
36					89
47					
5					
89					

	c	d	$	S	C
0	d36	d47		1	2
1					
2	d36	d47			5
36	d36	d47			89
47					
5					
89					

Analizadores sintácticos ascendentes

r_1. S' ::= S

r_2. S ::= C C

r_3. C ::= c C

r_4. C::= d

Los diferentes tipos de analizadores sintácticos ascendentes – LR (0), SLR, LALR, LR (1) y LR (k) – sólo se diferencian entre sí por el procedimiento de construcción de las tablas acción e ir-A. En esta sección abordaremos su estudio

Construcción de las tablas LALR

Ejemplo

	c	d	$	S	C
0	d36	d47		1	2
1					
2	d36	d47			5
36	d36	d47			89
47	r4	r4	r4		
5			r2		
89	r3	r3	r3		

	c	d	$	S	C
0	d36	d47	error	1	2
1	error	error	ok		
2	d36	d47	error		5
36	d36	d47	error		89
47	r4	r4	r4		
5	error	error	r2		
89	r3	r3	r3		

Análisis sintáctico ascendente

Los analizadores sintácticos ascendentes son capaces de decidir qué regla de producción aplicar a cada paso en función de los elementos terminales que se encuentran en la cabeza de lectura de la cadena de entrada. Como consecuencia se consigue un proceso de análisis con complejidad lineal O (n) con respecto al tamaño del problema. Estos analizadores son llamados genéricamente analizadores LR (K) o analizadores por reducción desplazamiento

Ejercicios

Una vez estudiados la construcción de las tablas para los 4 tipos de analizadores sintácticos ascendentes es posible comprobar su viabilidad con diferentes gramáticas

E ::= E + T | E – T | T
T ::= T * F | T / F | F
F ::= id | n | (E)

S ::= P
P ::= (P) P |

P ::= C L F
C ::= id ;
L ::= id ; L |
F ::= id (L) ;

P ::= L
L := S ; L |
S ::= E | B
B ::= { L }
E ::= id = E | n

Gestión de errores

La labor de un analizador sintáctico no se limita exclusivamente a reconocer que una frase pertenece al lenguaje generado por una gramática. Adicionalmente, en el caso de que existan errores sintácticos (o léxicos no reconocidos en la fase de análisis léxico), el analizador debe ser capaz de reconocer errores y emitir mensajes de información asociados

r_1. S' ::= S

r_2. S ::= C C

r_3. C ::= c C

r_4. C::= d

I. Identificación de errores

Cada entrada vacía de la tabla de acción de los analizadores ascendentes identifica una casuística diferente de error. En principio cada error debería incorporar un mensaje característico diferente

c	d	$
d3	d4	error
error	error	error
d6	d7	error
d3	d4	error
r4	r4	error
r2	r2	error
d6	d7	error
error	error	r4
r3	r3	error
error	error	r3

Terminación inesperada de la frase. Se esperaba c o d

Se encontró un c inesperado

II. Generación de mensajes

Cada celda de la tabla de acción permite identificar un tipo de error distinto y asociarle un mensaje de error diferente

Gestión de errores

La labor de un analizador sintáctico no se limita exclusivamente a reconocer que una frase pertenece al lenguaje generado por una gramática. Adicionalmente, en el caso de que existan errores sintácticos (o léxicos no reconocidos en la fase de análisis léxico), el analizador debe ser capaz de reconocer errores y emitir mensajes de información asociados

r_1. S' ::= S

r_2. S ::= C C

r_3. C ::= c C

r_4. C::= d

III. Localización de errores

Para reducir el número de mensajes de error podemos forzar reducciones en todos los estados donde solo se reduzca por una regla. Esto atrasa la emisión del error hasta el siguiente desplazamiento lo que puede complicar la localización

Las reducciones en rojo corresponden en realidad a situaciones de error

c	d	$
d3	d4	error
error	error	error
d6	d7	error
d3	d4	error
r4	r4	r4
r2	r2	r2
d6	d7	error
r4	r4	r4
r3	r3	r3
r3	r3	r3

Como heurística general nunca se deben consumir más de 3 tokens en modo de pánico

VI. Recuperación de errores

Se pasa a consumición de tokens en modo de pánico. Debe procurarse que el número de tokens consumido no sea muy elevado. Este proceso esta dirigido por conjuntos de predicción

Interpretación de conflictos en los tipos de analizadores

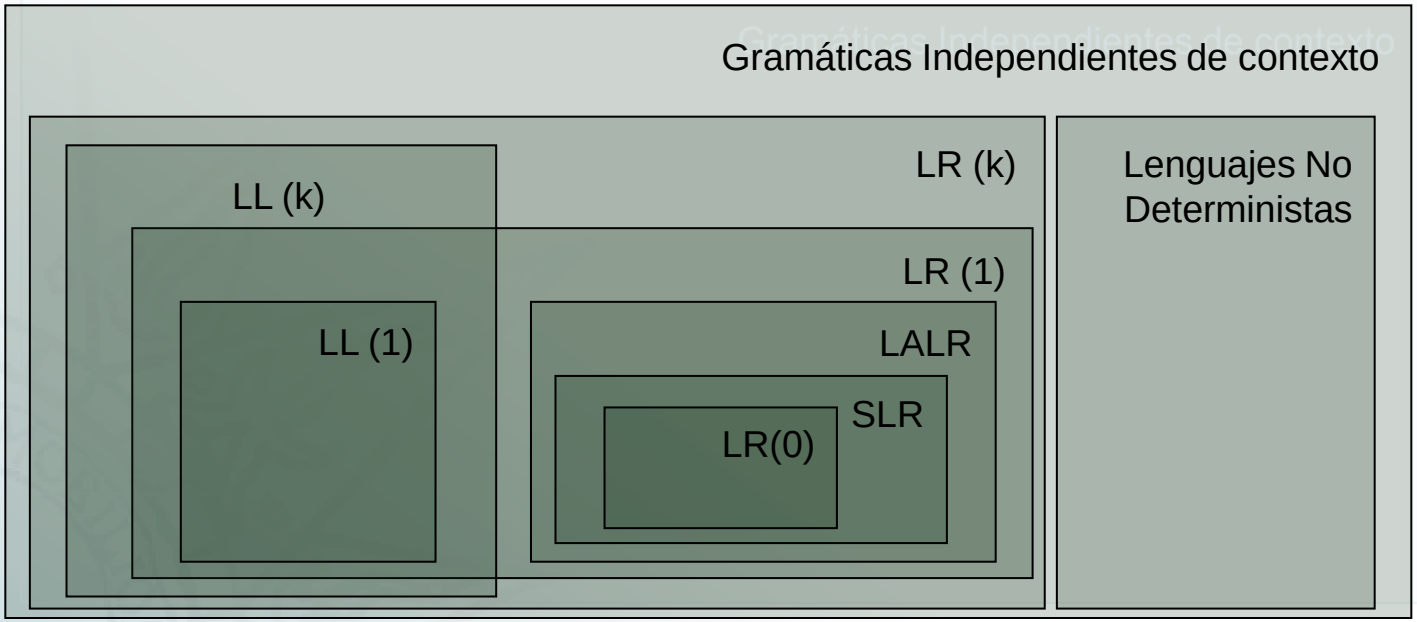
Como describimos al inicio del tema, existen dos tipos de conflictos que pueden darse dentro del análisis sintáctico ascendente por reducción desplazamiento. En este apartado discutiremos, para cada tipo gramática, qué condiciones debe verificar la tabla de acción de los analizadores para que no se den tales conflictos

- Conflictos reducción – reducción
- Conflictos reducción desplazamiento

	Reducción - Reducción	Reducción - Desplazamiento
Lenguajes LR (0)	No puede haber 2 reglas distintas en la misma fila de un estado de reducción	No puede haber acciones reducir y desplazar en una misma fila
Lenguajes SLR	No puede haber 2 reglas distintas en la misma celda de reducción	No puede haber una acción reducir y otra desplazar en una misma celda
Lenguajes LALR		
Lenguajes LALR		

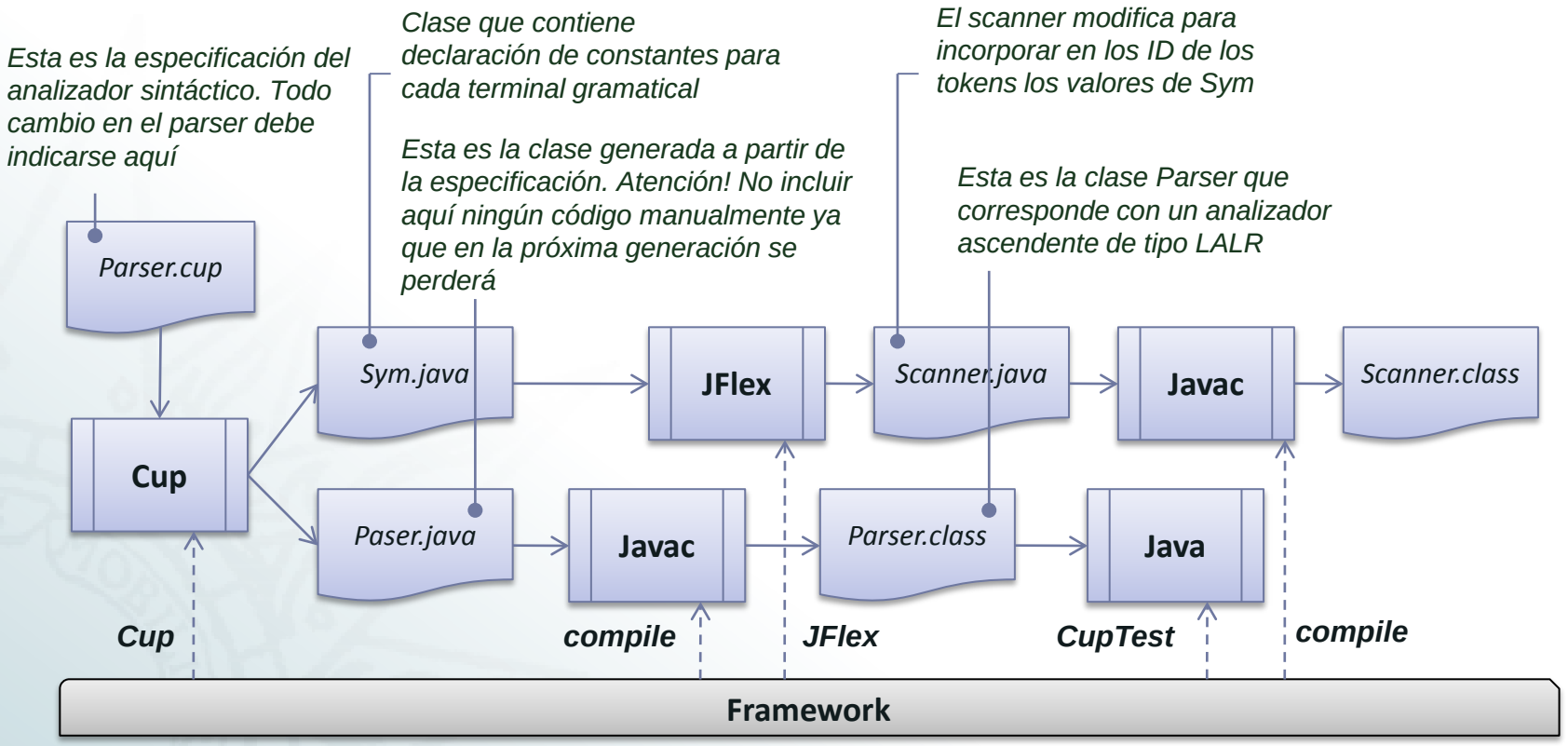
Clasificación de gramáticas por tipos de analizadores

Cuando la tabla de un analizador sintáctico de un determinado tipo no presenta conflictos de análisis se puede asegurar que la gramática del lenguaje bajo análisis es del tipo del analizador utilizado. Esto, en conjunción con la información extraída del tema anterior permite trazar la siguiente clasificación conjuntista de gramáticas independientes del contexto en función del tipo de análisis que soportan



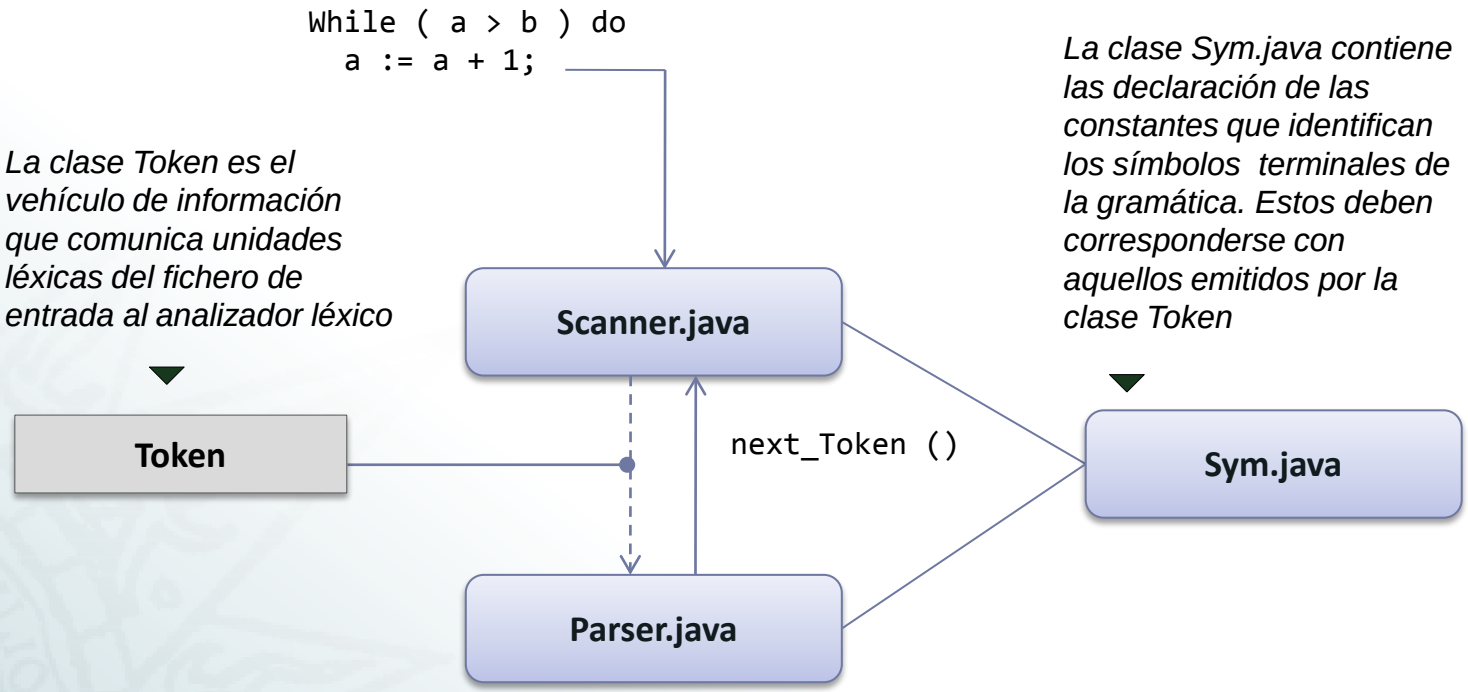
¿Qué es Cup?

Cup es una herramienta para la generación de analizadores sintácticos escritos en java. A partir de un fichero de especificación que describe las características sintácticas de un lenguaje, Cup genera un código fuente compilable que puede ser utilizado como analizador sintáctico



¿Qué es Cup?

Cup es una herramienta para la generación de analizadores sintácticos escritos en java. A partir de un fichero de especificación que describe las características sintácticas de un lenguaje, Cup genera un código fuente compilable que puede ser utilizado como analizador sintáctico



¿Cómo funciona Cup?

La especificación de un analizador sintáctico en cup está compuesta por varias secciones diferentes. El marco de trabajo tecnológico utilizado para el desarrollo del compilador proporciona una plantilla de tal especificación que debe ser extendida pero NO modificada (seguir comentarios)

I. Paquetes e importaciones

Se incluye la declaración de todas clases y paquetes necesarias para compilar adecuadamente el analizador sintáctico. Ninguna de las importaciones de la plantilla debe ser eliminada!

```
4  import java_cup.runtime.Symbol;
5  import java.util.*;
6  import es.uned.compiler.lexical.*;
7  import es.uned.compiler.code.*;
8  import es.uned.compiler.intermediate.*;
9  import es.uned.compiler.semantic.*;
10 import es.uned.compiler.semantic.symbol.*;
11 import es.uned.compiler.semantic.type.*;
12 import es.uned.compiler.syntax.*;
13 import compiler.CompilerContext;
14 import compiler.lexical.*;
15 import compiler.syntax.nonTerminal.*;
16 import compiler.semantic.*;
17 import compiler.semantic.symbol.*;
18 import compiler.semantic.type.*;
19 import compiler.intermediate.*;
20 import compiler.code.*;
21
```

Consúltese el manual de usuario de Cup

```
1  package compiler.syntax;
2
3  // Declaración de importaciones
4  import java_cup.runtime.Symbol;
5  import java.util.*;
6  import es.uned.compiler.lexical.*;
7  import es.uned.compiler.code.*;
8  import es.uned.compiler.intermediate.*;
9  import es.uned.compiler.semantic.*;
10 import es.uned.compiler.semantic.symbol.*;
11 import es.uned.compiler.semantic.type.*;
12 import es.uned.compiler.syntax.*;
13 import compiler.CompilerContext;
14 import compiler.lexical.*;
15 import compiler.syntax.nonTerminal.*;
16 import compiler.semantic.*;
17 import compiler.semantic.symbol.*;
18 import compiler.semantic.type.*;
19 import compiler.intermediate.*;
20 import compiler.code.*;
21
22 // Declaración del código de usuario
23 action code {:
24
25     SyntaxErrorManager   syntaxErrorManager   = CompilerContext.
26     SemanticErrorManager semanticErrorManager = CompilerContext.
27     ScopeManagerIF       scopeManager         = CompilerContext.
28     FinalCodeFactoryIF   finalCodeFactory     = CompilerContext.
29
30 :}
31
32 parser code {:
33     SyntaxErrorManager syntaxErrorManager = CompilerContext.getS
34
35     public void syntax_error(Symbol symbol)
36     {
37         Token token = (Token) symbol.value;
38         syntaxErrorManager.syntaxError ("Error sintáctico", toke
39     }
40
41     public void unrecovered_syntax_error(java_cup.runtime.Symbol
42     {
43         Token token = (Token) symbol.value;
44         syntaxErrorManager.syntaxFatalError ("Error fatal", toke
45     }
46 :}
```

¿Cómo funciona Cup?

La especificación de un analizador sintáctico en cup está compuesta por varias secciones diferentes. El marco de trabajo tecnológico utilizado para el desarrollo del compilador proporciona una plantilla de tal especificación que debe ser extendida pero NO modificada (seguir comentarios)

II. Action code

Se incluyen declaraciones de variables y métodos auxiliares que serán utilizados dentro de la gramática. Algunas de las declaraciones presentes en la plantilla son necesarias solamente de cara al segundo cuatrimestre. Si las elimina o comenta deberá recuperarlas en la segunda entrega. En el primer cuatrimestres sólo SyntaxErrorManager es utilizado

```
23  action code {:
24
25      SyntaxErrorManager   syntaxErrorManager   = CompilerContext.getSyntaxErrorManager();
26      SemanticErrorManager semanticErrorManager = CompilerContext.getSemanticErrorManager ();
27      ScopeManagerIF       scopeManager         = CompilerContext.getScopeManager ();
28      FinalCodeFactoryIF   finalCodeFactory     = CompilerContext.getFinalCodeFactory ();
29
30  :}
```

Consúltese el manual de usuario de Cup

```
1   package compiler.syntax;
2
3   // Declaración de importaciones
4   import java_cup.runtime.Symbol;
5   import java.util.*;
6   import es.uned.compiler.lexical.*;
7   import es.uned.compiler.code.*;
8   import es.uned.compiler.intermediate.*;
9   import es.uned.compiler.semantic.*;
10  import es.uned.compiler.semantic.symbol.*;
11  import es.uned.compiler.semantic.type.*;
12  import es.uned.compiler.syntax.*;
13  import compiler.CompilerContext;
14  import compiler.lexical.*;
15  import compiler.syntax.nonTerminal.*;
16  import compiler.semantic.*;
17  import compiler.semantic.symbol.*;
18  import compiler.semantic.type.*;
19  import compiler.intermediate.*;
20  import compiler.code.*;
21
22  // Declaración del código de usuario
23  action code {:
24
25      SyntaxErrorManager   syntaxErrorManager   = CompilerContext.
26      SemanticErrorManager semanticErrorManager = CompilerContext.
27      ScopeManagerIF       scopeManager         = CompilerContext.
28      FinalCodeFactoryIF   finalCodeFactory     = CompilerContext.
29
30  :}
31
32  parser code {:
33      SyntaxErrorManager syntaxErrorManager = CompilerContext.getS
34
35      public void syntax_error(Symbol symbol)
36      {
37          Token token = (Token) symbol.value;
38          syntaxErrorManager.syntaxError ("Error sintáctico", toke
39      }
40
41      public void unrecovered_syntax_error(java_cup.runtime.Symbol
42      {
43          Token token = (Token) symbol.value;
44          syntaxErrorManager.syntaxFatalError ("Error fatal", toke
45      }
46  :}
```

¿Cómo funciona Cup?

La especificación de un analizador sintáctico en cup está compuesta por varias secciones diferentes. El marco de trabajo tecnológico utilizado para el desarrollo del compilador proporciona una plantilla de tal especificación que debe ser extendida pero NO modificada (seguir comentarios)

III. Parser code

Permite incluir declaraciones de variables y métodos que ayudan a adaptar el comportamiento del analizador. La plantilla incluye la declaración del SyntaxErrorManager y funciones manejadoras de los errores recuperables y no recuperables. Ninguna de estas declaraciones deben ser eliminadas

```
32  parser code {:
33      SyntaxErrorManager syntaxErrorManager = CompilerContext.getSyntaxErrorManager();
34
35      public void syntax_error(Symbol symbol)
36      {
37          Token token = (Token) symbol.value;
38          syntaxErrorManager.syntaxError ("Error sintáctico", token);
39      }
40
41      public void unrecovered_syntax_error(java_cup.runtime.Symbol symbol)
42      {
43          Token token = (Token) symbol.value;
44          syntaxErrorManager.syntaxFatalError ("Error fatal", token);
45      }
46  :}
```

Consúltese el manual de usuario de Cup

```
1   package compiler.syntax;
2
3   // Declaración de importaciones
4   import java_cup.runtime.Symbol;
5   import java.util.*;
6   import es.uned.compiler.lexical.*;
7   import es.uned.compiler.code.*;
8   import es.uned.compiler.intermediate.*;
9   import es.uned.compiler.semantic.*;
10  import es.uned.compiler.semantic.symbol.*;
11  import es.uned.compiler.semantic.type.*;
12  import es.uned.compiler.syntax.*;
13  import compiler.CompilerContext;
14  import compiler.lexical.*;
15  import compiler.syntax.nonTerminal.*;
16  import compiler.semantic.*;
17  import compiler.semantic.symbol.*;
18  import compiler.semantic.type.*;
19  import compiler.intermediate.*;
20  import compiler.code.*;
21
22  // Declaración del código de usuario
23  action code {:
24
25      SyntaxErrorManager   syntaxErrorManager   = CompilerContext.
26      SemanticErrorManager semanticErrorManager = CompilerContext.
27      ScopeManagerIF       scopeManager         = CompilerContext.
28      FinalCodeFactoryIF   finalCodeFactory     = CompilerContext.
29
30  :}
31
32  parser code {:
33      SyntaxErrorManager syntaxErrorManager = CompilerContext.getS
34
35      public void syntax_error(Symbol symbol)
36      {
37          Token token = (Token) symbol.value;
38          syntaxErrorManager.syntaxError ("Error sintáctico", toke
39      }
40
41      public void unrecovered_syntax_error(java_cup.runtime.Symbol
42      {
43          Token token = (Token) symbol.value;
44          syntaxErrorManager.syntaxFatalError ("Error fatal", toke
45      }
46  :}
```

¿Cómo funciona Cup?

La especificación de un analizador sintáctico en cup está compuesta por varias secciones diferentes. El marco de trabajo tecnológico utilizado para el desarrollo del compilador proporciona una plantilla de tal especificación que debe ser extendida pero NO modificada (seguir comentarios)

IV. Declaración de terminales

Se deben declarar extensivamente todos los símbolos terminales utilizados en la gramática. Se comienza por la palabra reservada terminal seguida de la clase java que representa el terminal y finalmente el nombre del terminal seguido de punto y coma (;)

Palabra reservada *terminal*

Clase que tipifica el Token

Nombre del Token

```
48  // Declaración de terminales (Ejemplo)
49  terminal Token PLUS;
50  terminal Token MINUS;
51  // ...
52
```

- *Todos y cada uno de los terminales definidos aquí aparecerán en la clase Sym para que puedan referirse desde el escáner*

```
48  // Declaración de terminales (Ejemplo)
49  terminal Token PLUS;
50  terminal Token MINUS;
51  // ...
52
53  // Declaración de no terminales
54  non terminal            program;
55  non terminal Axiom      axiom;
56  // ...
57
58  // Declaración de relaciones de precedencia
59  precedence left     PLUS, MINUS;
60  precedence left     MUL, DIV;
61  // ...
62
63  // Declaración de reglas de producción
64  start with program;
65
66  program ::=
67    {: syntaxErrorManager.syntaxInfo ("Starting parsing..."); :}
68    axiom:ax
69    {:
70          List intermediateCode = ax.getIntermediateCode ();
71          finalCodeFactory.create (intermediateCode);
72          syntaxErrorManager.syntaxInfo ("Parsing proccess ended.");
73    :};
```

¿Cómo funciona Cup?

Consúltese el manual de usuario de Cup

La especificación de un analizador sintáctico en cup está compuesta por varias secciones diferentes. El marco de trabajo tecnológico utilizado para el desarrollo del compilador proporciona una plantilla de tal especificación que debe ser extendida pero NO modificada (seguir comentarios)

V. Declaración de no terminales

Se deben declarar extensivamente todos los símbolos no terminales utilizados en las partes izquierda de las reglas de la gramática. Se comienza por la palabra reservada terminal seguida de la clase java que representa el no terminal y finalmente el nombre del no terminal seguido de punto y coma (;)

Palabra reservada non *terminal*

Clase que tipifica el no terminal

Nombre del no terminal

```
53  // Declaración de no terminales
54  non terminal            program;
55  non terminal Axiom      axiom;
56  // ...
57
```

- *La tipificación de los no terminales puede omitirse y postergarse hasta el segundo cuatrimestre*

```
48  // Declaración de terminales (Ejemplo)
49  terminal Token PLUS;
50  terminal Token MINUS;
51  // ...
52
53  // Declaración de no terminales
54  non terminal            program;
55  non terminal Axiom      axiom;
56  // ...
57
58  // Declaración de relaciones de precedencia
59  precedence left     PLUS, MINUS;
60  precedence left     MUL, DIV;
61  // ...
62
63  // Declaración de reglas de producción
64  start with program;
65
66  program ::=
67    {: syntaxErrorManager.syntaxInfo ("Starting parsing..."); :}
68    axiom:ax
69    {:
70          List intermediateCode = ax.getIntermediateCode ();
71          finalCodeFactory.create (intermediateCode);
72          syntaxErrorManager.syntaxInfo ("Parsing proccess ended.");
73     :};
```

¿Cómo funciona Cup?

Consúltese el manual de usuario de Cup

La especificación de un analizador sintáctico en cup está compuesta por varias secciones diferentes. El marco de trabajo tecnológico utilizado para el desarrollo del compilador proporciona una plantilla de tal especificación que debe ser extendida pero NO modificada (seguir comentarios)

VI. Declaración explícita de precedencia y asociatibidad

En Cup las gramáticas de operadores pueden ser ambiguas. La resolución de la ambigüedad se realiza de forma explícita a partir de la información de precedencia y asociatividad

Igual precedencia →

Precedencia creciente ↓

Palabra reservada precedence

Información de asociatividad (left | right | nonassoc)

Nombre del terminal

```
58  // Declaración de relaciones de precedencia
59  precedence left     PLUS, MINUS;
60  precedence left     MUL, DIV;
61  // ...
62
```

- *La tipificación de los no terminales puede omitirse y postergarse hasta el segundo cuatrimestre*

```
48  // Declaración de terminales (Ejemplo)
49  terminal Token PLUS;
50  terminal Token MINUS;
51  // ...
52
53  // Declaración de no terminales
54  non terminal            program;
55  non terminal Axiom      axiom;
56  // ...
57
58  // Declaración de relaciones de precedencia
59  precedence left     PLUS, MINUS;
60  precedence left     MUL, DIV;
61  // ...
62
63  // Declaración de reglas de producción
64  start with program;
65
66  program ::=
67    {: syntaxErrorManager.syntaxInfo ("Starting parsing..."); :}
68    axiom:ax
69    {:
70          List intermediateCode = ax.getIntermediateCode ();
71          finalCodeFactory.create (intermediateCode);
72          syntaxErrorManager.syntaxInfo ("Parsing proccess ended.");
73     :};
```

¿Cómo funciona Cup?

Consúltese el manual de usuario de Cup

La especificación de un analizador sintáctico en cup está compuesta por varias secciones diferentes. El marco de trabajo tecnológico utilizado para el desarrollo del compilador proporciona una plantilla de tal especificación que debe ser extendida pero NO modificada (seguir comentarios)

VII. Declaración del axioma

En Cup es posible declarar el axioma de la gramática. Esta declaración es opcional. Si no aparece se asume por defecto que el axioma corresponde con el antecedente de la primera regla de producción. No borre esta declaración

Palabra reservada start with

Nombre del no terminal axioma

```
62
63  // Declaración de reglas de producción
64  start with program;
65
```

```
48  // Declaración de terminales (Ejemplo)
49  terminal Token PLUS;
50  terminal Token MINUS;
51  // ...
52
53  // Declaración de no terminales
54  non terminal            program;
55  non terminal Axiom      axiom;
56  // ...
57
58  // Declaración de relaciones de precedencia
59  precedence left     PLUS, MINUS;
60  precedence left     MUL, DIV;
61  // ...
62
63  // Declaración de reglas de producción
64  start with program;
65
66  program ::=
67    {: syntaxErrorManager.syntaxInfo ("Starting parsing..."); :}
68    axiom:ax
69    {:
70          List intermediateCode = ax.getIntermediateCode ();
71          finalCodeFactory.create (intermediateCode);
72          syntaxErrorManager.syntaxInfo ("Parsing proccess ended.");
73     :};
```

¿Cómo funciona Cup?

Consúltese el manual de usuario de Cup

La especificación de un analizador sintáctico en cup está compuesta por varias secciones diferentes. El marco de trabajo tecnológico utilizado para el desarrollo del compilador proporciona una plantilla de tal especificación que debe ser extendida pero NO modificada (seguir comentarios)

VIII. Declaración del reglas de producción

La última sección permite declarar las reglas de producción gramatical. Los no terminales van en minúscula, los terminales en mayúscula y el símbolo de producción es ::=

Antecedente

Consecuente

```
65
66  program ::=
67    {: syntaxErrorManager.syntaxInfo ("Starting parsing..."); :}
68    axiom:ax
69    {:
70          List intermediateCode = ax.getIntermediateCode ();
71          finalCodeFactory.create (intermediateCode);
72          syntaxErrorManager.syntaxInfo ("Parsing proccess ended.");
73     :};
```

- *La parte derecha incluye acciones semánticas . Se usarán en el segundo cuatrimestre. No borre las acciones de la plantilla*

```
48  // Declaración de terminales (Ejemplo)
49  terminal Token PLUS;
50  terminal Token MINUS;
51  // ...
52
53  // Declaración de no terminales
54  non terminal            program;
55  non terminal Axiom      axiom;
56  // ...
57
58  // Declaración de relaciones de precedencia
59  precedence left     PLUS, MINUS;
60  precedence left     MUL, DIV;
61  // ...
62
63  // Declaración de reglas de producción
64  start with program;
65
66  program ::=
67    {: syntaxErrorManager.syntaxInfo ("Starting parsing..."); :}
68    axiom:ax
69    {:
70          List intermediateCode = ax.getIntermediateCode ();
71          finalCodeFactory.create (intermediateCode);
72          syntaxErrorManager.syntaxInfo ("Parsing proccess ended.");
73     :};
```

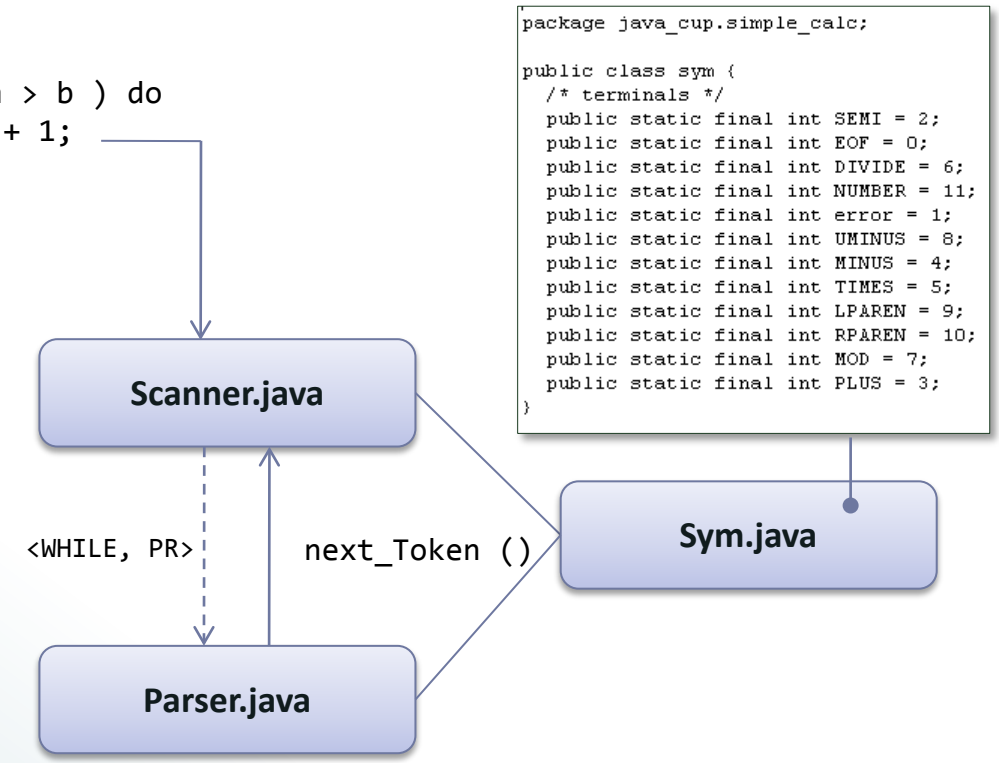
¿Cómo se usa Cup?

> **Consúltese el manual de usuario de Cup y JFlex**

El analizador sintáctico generado por Cup debe integrarse con el analizador léxico generado por JFlex. Ambas herramientas están preparadas para integrarse fácilmente. Además el marco de trabajo donde se desarrolla el compilador garantiza dicha integración

Lo único que el usuario debe asegurar es que los identificadores de los tokens generados por el escaner correspondan con constantes declaradas en la clase Sym. Síganse los siguientes pasos:

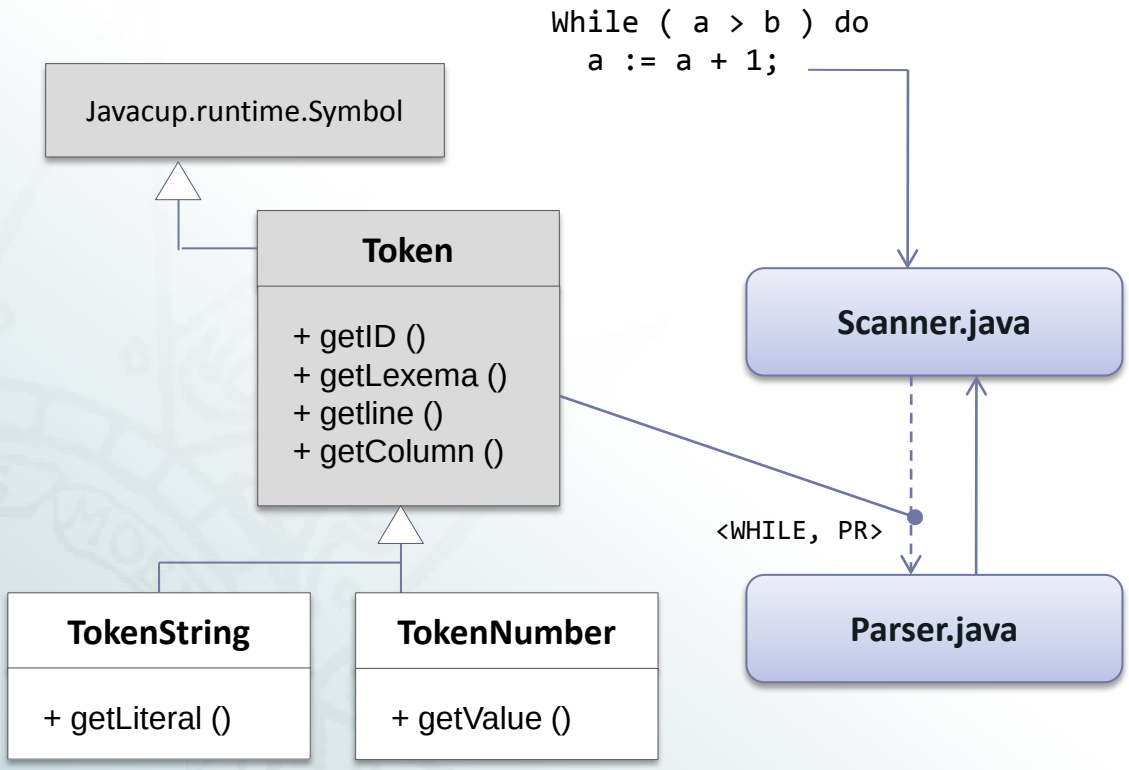
1. Crear escaner con ID de token falsos
2. Crear el analizador sintáctico
    1. Declarar terminales
    2. Declara no terminales
    3. Declarar reglas…
    4. General el parser invocando a Cup
3. Reconstruir escáner
    1. Sustituir los ID falsos por los de Sym
    2. Generar el escaner invocando a JFlex



¿Cómo se usa Cup?

El analizador sintáctico generado por Cup debe integrarse con el analizador léxico generado por JFlex. Ambas herramientas están preparadas para integrarse fácilmente. Además el marco de trabajo donde se desarrolla el compilador garantiza dicha integración

Consúltese el manual de usuario de Cup y JFlex



Cada vez que el escáner reconoce un nuevo patrón léxico a la entrada genera un token y lo entrega al parser. El framework proporciona la clase Token para vehicular esta entrega

1. Especificar el escáner en JFlex
2. Para cada patrón léxico
    1. Emitir el tipo de token apropiado
    2. Enriquecer el token con datos
3. Si el tipo de token es especial
    1. Implementar una clase nueva
    2. Devolver una instancia de ella

Gestión de errores en Cup

Consúltese el manual de usuario de Cup

Cuando se realiza un análisis ascendente en Cup de una cadena de entrada es posible que el parser generado encuentre dos posibles situaciones de error

I. Errores recuperables

Los errores recuperables son aquellos de los que el compilador es capaz de escaparse en estado de pánico y continuar después en un contexto gramatical estable. Se describen con la producción error al final de la gramática

```
exp::= exp ADD exp |
       exp SUB exp |
       exp MUL exp |
       exp DIV exp |
       NUMBER      |
       error;
```

```
exp::= exp ADD exp |
       exp SUB exp |
       exp MUL exp |
       exp DIV exp |
       NUMBER      |
       error PYC   |
       error PI;
```

Cuando la cadena a la entrada no encaja con ninguna de las partes derechas de las reglas de exp se reduce por las reglas de error. Estas deben ir normativamente las últimas

```
32  parser code {:
33      SyntaxErrorManager syntaxErrorManager = CompilerContext.getSyntaxErrorManager();
34
35      public void syntax_error(Symbol symbol)
36      {
37          Token token = (Token) symbol.value;
38          syntaxErrorManager.syntaxError ("Error sintáctico", token);
39      }
40
41      public void unrecovered_syntax_error(java_cup.runtime.Symbol symbol)
42      {
43          Token token = (Token) symbol.value;
44          syntaxErrorManager.syntaxFatalError ("Error fatal", token);
45      }
46  :}
```

Cada vez que se reduce por una producción de error, Cup invoca automáticamente al manejador de errores recuperables syntax_error. En la plantilla del framework el tratamiento consiste en emitir un mensaje de error con el SyntaxErrorManager . No altere este código

Gestión de errores en Cup

Consúltese el manual de usuario de Cup

Cuando se realiza un análisis ascendente en Cup de una cadena de entrada es posible que el parser generado encuentre dos posibles situaciones de error

II. Errores irrecuperables

Los errores irrecuperables son aquellos de los que el compilador no es capaz de recuperarse una vez que han sido encontrados y provocan que el parser aborte el proceso de análisis sintáctico

```
32  parser code {:
33      SyntaxErrorManager syntaxErrorManager = CompilerContext.getSyntaxErrorManager();
34
35      public void syntax_error(Symbol symbol)
36      {
37          Token token = (Token) symbol.value;
38          syntaxErrorManager.syntaxError ("Error sintáctico", token);
39      }
40
41      public void unrecovered_syntax_error(java_cup.runtime.Symbol symbol)
42      {
43          Token token = (Token) symbol.value;
44          syntaxErrorManager.syntaxFatalError ("Error fatal", token);
45      }
46  :}
```

Cada vez que se produce un error irrecuperable, Cup invoca automáticamente al manejador de errores recuperables unrecovered_syntax_error. En la plantilla del framework el tratamiento consiste en emitir un mensaje de error fatal con el SyntaxErrorManager . No altere este código

Gestión de errores en Cup

Consúltese el manual de usuario de Cup

Cuando se realiza un análisis ascendente en Cup de una cadena de entrada es posible que el parser generado encuentre dos posibles situaciones de error

Traza de errores

Además es posible trazar los error del compilador, o la ejecución del mismo a lo largo del proceso ascendente mediante la inserción de acciones en las partes derechas de la regla que invoquen el gestor de errores sintácticos proporcionado por el framework de desarrollo

SyntaxErrorManager

+ syntaxDebug (String message)
+ syntaxInfo (String message)
+ syntaxWarn (String message)
+ SyntaxError (String message)
+ SyntaxFatal (String message)

...

```
exp ::= exp MAS exp {: syntaxErrorManager.syntaxDebug (“sumando”);       :}
      | exp POR exp {: syntaxErrorManager.syntaxDebug (“multiplicando”); :}
      | error       {: syntaxErrorManager.syntaxDebug (“error”);         :}
```

Desarrollo paso a paso

El proceso de desarrollo del analizador sintáctico puede resumirse en una serie de pasos que pasamos a describir detalladamente a continuación

I. Análisis del lenguaje

Lo primero es analizar la estructura sintáctica del lenguaje para el cual tratamos de generar un analizador sintáctico. Para este proceso recomendamos los siguientes pasos

1. Seleccionar un código fuente representativo (completo)
2. Organizarlo estructuralmente en bloques sintácticos
3. Nombrar simbólicamente a cada bloque (con un no terminal)

Programa
Declaraciones
DeclaracionConstantes
DeclaracionConstante
DeclaraciónTipos
DeclaracionTipo
DeclaraciónRegistro
DeclaracionVariable…

DeclaraciónSubprogramas
DeclaraciónSubprogrograma
DeclaraciónProcedimiento
DeclaraciónFunción
DeclararaciónCabecera
DeclaraciónCuerpo…

Sentencia
ListaSentencias
BloqueSentencias
SentenciaWhile…

```
Program Prueba;

Const PI = 3.14
Type TPunto = RECORD
       x, y : INTEGER;
      END
      TCirculo = RECORD
        centro : TPunto;
        radio  : REAL;
      END;
Var x, y : REAL;
    c    : TCirculo;

PROCEDURE crear (p: TPunto,
                 r: REAL,
                 VAR c:TCirculo)
BEGIN
   c.centro := p;
   c.radio  := r
END;

FUNCTION area (c: TCirculo): REAL
  VAR radio: REAL;
  FUNCTION potencia (r:REAL, n:INTEGER): REAL
  BEGIN
    ...
  END;
BEGIN
   radio := c.radio;
   area  := PI * Potencia (radio, 2);
END
BEGIN ... END;
```

Desarrollo paso a paso

El proceso de desarrollo del analizador sintáctico puede resumirse en una serie de pasos que pasamos a describir detalladamente a continuación

II. Especificación gramatical

El siguiente paso es realizar la especificación gramatical del lenguaje dentro de Cup.

1. Declare conjunto de no terminales
2. Declare conjunto de terminales
3. Escribir reglas de producción

Solo deben existir en la especificación estos dos tipos de reglas. Nunca escriba una regla mezcla de ambos tipos

Reglas de refinamiento

Las reglas de refinamiento se encargar de redefinir una abstracción sintáctica – representada por un no terminal – en un conjunto de terminales y no terminales

```
Declaraciones ::= DeclaraciónContantes |
                  DeclaraciónTipos     |
                  DeclaraciónVariables;

DeclaraciónVariables ::= VAR ListaDeclaraciónVariables;
```

Reglas de resolución de recursividad

Las reglas de resolución de recursividad se encargan de establecer las definiciones recursivas de los terminales de refinamiento

```
ListaDeclaraciónVariables ::= DeclaraciónVariable |
   ListaDeclaraciónVariables DeclaraciónVariable;

ListaParamActuales ::= exp COMA ListaParamActuales |
                       ;
```

Desarrollo paso a paso

El proceso de desarrollo del analizador sintáctico puede resumirse en una serie de pasos que pasamos a describir detalladamente a continuación

III. Generación y prueba del parser

El tercer y ultimo paso consiste en generar el analizador sintáctico y probarlo

1. Genere el analizador sintáctico con Cup
    1. Ejecute Clean
    2. Ejecute Cup
    3. Ejecute compile

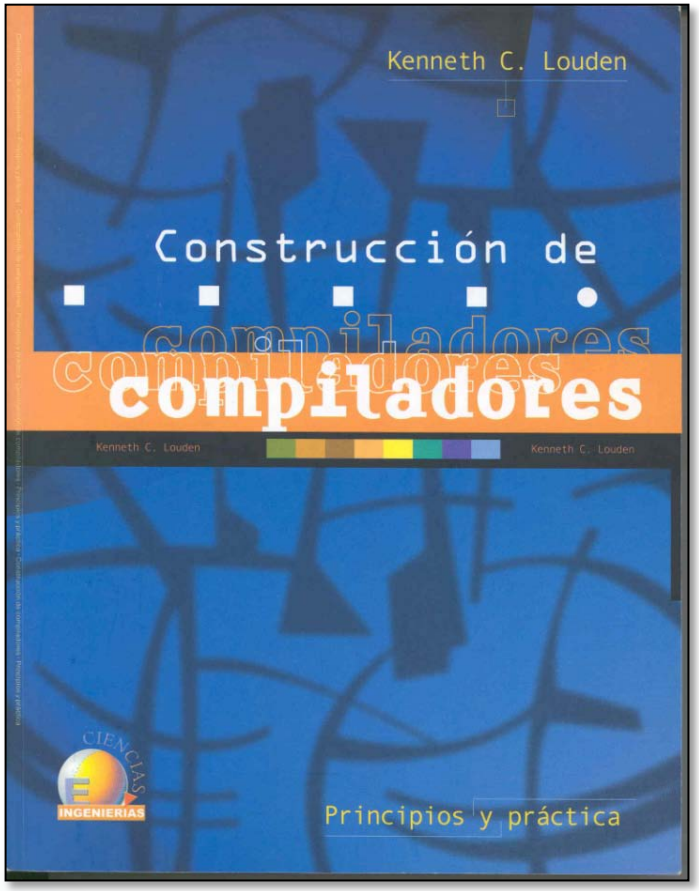
Alternativamente el framework también proporciona la tarea build que ejecuta clean + cup + flex + compile

2. Adapte la especificación de JFlex
    1. Importe la clase Sym en JFlex
    2. Cambie las acciones de JFlex para generar tokens con ID en Sym
    3. Ejecute jflex para generar de nuevo el escáner
3. Pruebe el parser
    1. Ejecute cupTest
    2. Cambie en el build.xml el fichero fuente de entrada para probar distintos casos

Material de estudio

Bibliografía básica

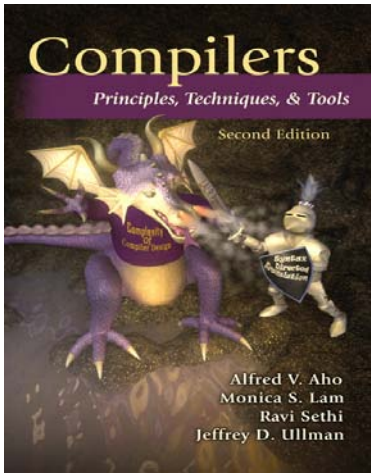
Construcción de compiladores: principios y práctica
Kenneth C. Louden International Thomson Editores, 2004 ISBN 970-686-299-4



Material de estudio

Bibliografía complementaria

Compiladores: Principios, técnicas y herramientas.
Segunda Edición Aho, Lam, Sethi, Ullman
Addison – Wesley, Pearson Educación, México 2008



Diseño de compiladores. A. Garrido, J. Iñesta, F. Moreno y J. Pérez. 2002. Edita Universidad de Alicante