uc3m | Universidad Carlos III de Madrid
Departamento de Informática

Tratamiento de Ambigüedad y Error

1

Ambigüedad

- Para esta sentencia:

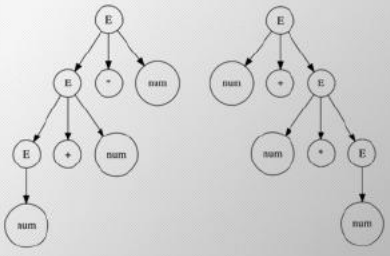
  num + num * num

- Con la gramática:

$$E \longrightarrow E + E$$
$$E \longrightarrow E * E$$
$$E \longrightarrow num$$

- Se puede aplicar diferentes derivaciones que la reconozcan, dando lugar a diferentes árboles gramaticales

- No hay un algoritmo que determine si una gramática es ambigua
- Por tanto, se van a estudiar casos prácticos concretos y reglas que pueden solucionar la ambigüedad en esas circunstancias



Ambigüedad

Solución

- Se puede eliminar la ambigüedad:
  - Alterando la gramática
    - Se obtiene una gramática equivalente, que reconoce el mismo lenguaje pero sin ambigüedades
  - De forma explícita
    - Se mantiene la gramática ambigua pero se toman decisiones de análisis que resuelven la recursividad
  - Ambigüedad no esencial
    - Ambigüedades que no suponen un problema para el análisis de las expresiones

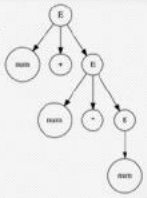
Ambigüedad: Alteración de la gramática

Gramática de operadores

- Sea la gramática:

$E \longrightarrow num + E$
$E \longrightarrow num * E$
$E \longrightarrow num$

Recursiva a derecha



- Si la entrada es:

num + num * num

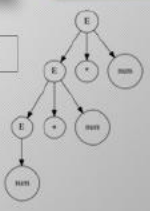
- El análisis ascendente sería:

Entrada	Pila	Acción
num + ***num*** * ***num*** $	0 $	*D*2
+ ***num*** * ***num*** $	2 ***num*** 0 $	*D*3
num * ***num*** $	3 + 2 ***num*** 0 $	*D*2
* ***num*** $	2 ***num*** 3 + 2 ***num*** 0 $	*D*4
num $	4 * 2 ***num*** 3 + 2 ***num*** 0 $	*D*2
$	2 ***num*** 4 * 2 ***num*** 3 + 2 ***num*** 0 $	*R*(*E* ⟶ ***num***)
$	6 *E* 4 * 2 ***num*** 3 + 2 ***num*** 0 $	*R*(*E* ⟶ ***num*** * *E*)
$	5 *E* 3 + 2 ***num*** 0 $	*R*(*E* ⟶ ***num*** + *E*)
$	1 *E* 0 $	ACP

- Sea la gramática:

$E \longrightarrow E + num$
$E \longrightarrow E * num$
$E \longrightarrow num$

Recursiva a izquierda



- Si la entrada es:

num + num * num

- El análisis ascendente sería:

Entrada	Pila	Acción
num + ***num*** * ***num*** $	0 $	*D*2
+ ***num*** * ***num*** $	2 ***num*** 0 $	*R*(*E* ⟶ ***num***)
+ ***num*** * ***num*** $	1 *E* 0 $	*D*3
num * ***num*** $	3 + 1 *E* 0 $	*D*5
* ***num*** $	5 ***num*** 3 + 1 *E* 0 $	*R*(*E* ⟶ ***num*** + *E*)
* ***num*** $	1 *E* 0 $	*D*4
num $	4 * 1 *E* 0 $	*D*6
$	6 ***num*** 4 * 1 *E* 0 $	*R*(*E* ⟶ ***num*** * *E*)
$	1*E* 0 $	ACP

Ambigüedad: Alteración de la gramática

Asociatividad

- Con la gramática Recursiva a derecha:

  $E \longrightarrow num + E$
  $E \longrightarrow num * E$
  $E \longrightarrow num$

- Se evalúa haciendo asociatividad a derecha:

  $num + (num * num)$

- Con la gramática Recursiva a izquierda:

  $E \longrightarrow E + num$
  $E \longrightarrow E * num$
  $E \longrightarrow num$

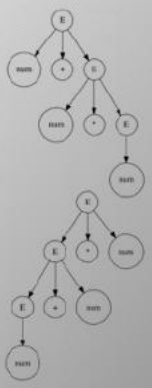
- Se evalúa haciendo asociatividad a izquierda:

  $(num + num) * num$

- Si la expresión tiene los mismos operadores entonces habría que tomar la forma recursiva a izquierdas, ya que es la manera habitual de evaluación matemática
- Pero diferentes operadores pueden alterar el orden de evaluación esto es:

- precedencia +	Operadores
	() - unario
	* /
	+ - binario

- ¿Cómo puede resolverse la precedencia?
  - ¿Qué nivel se resuelve antes el k o el $k+1$?
  - Resp: Los niveles más profundos antes



Ambigüedad: Alteración de la gramática

Precedencia

State	*	+	num	$	E	F	T
0			D4		1	3	2
1		D6		ACP			
2	D6	R2	R2				
3	R4	R4	R4				
4	R5	R5	R5				
5			D4		3	7	
6			D4		8		
7	D6	R1	R1				
8	R3	R3	R3				

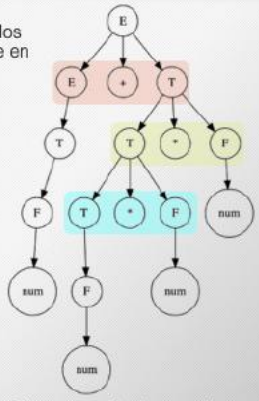
- Se modifican las reglas de producción para que los operadores de mayor precedencia estén siempre en un nivel más profundo

$E \longrightarrow E + num$
$E \longrightarrow E * num$
$E \longrightarrow num$

- La gramática transformada sería:

1) $E \longrightarrow E + T$
2) $E \longrightarrow T$
3) $T \longrightarrow T * F$
4) $T \longrightarrow F$
5) $F \longrightarrow num$

- La evaluación de la sentencia sería:
  num + num * num * num



Correspondería a evaluar:
num+((num*num)*num)

Entrada	Pila	Acción
num + ***num*** * ***num*** * ***num***$	0 $	*D*4
+ ***num*** * ***num**** ***num*** $	4 ***num*** 0 $	*R*(*F* → ***num***)
+ ***num*** * ***num**** ***num*** $	3 *F* 0 $	*R*(*T* → *F*)
+ ***num*** * ***num**** ***num*** $	2 *T* 0 $	*R*(*E* → *T*)
+ ***num*** * ***num**** ***num*** $	1 *E* 0 $	*D*5
num * ***num**** ***num*** $	5 + 1 *E* 0 $	*D*4
* ***num**** ***num*** $	4 ***num*** 5 + 1 *E* 0 $	*R*(*F* → ***num***)
* ***num**** ***num*** $	7 *T* 5 + 1 *E* 0 $	*D*6
num* ***num*** $	6 * 7 *T* 5 + 1 *E* 0 $	*D*4
* ***num*** $	4 ***num*** 6 * 7 *T* 5 + 1 *E* 0 $	*R*(*F* → ***num***)
* ***num*** $	8 *F* 6 * 7 *T* 5 + 1 *E* 0 $	*R*(*T* → *T* * *F*)
* ***num*** $	7 *T* 5 + 1 *E* 0 $	*D*6
num $	6 * 7 *T* 5 + 1 *E* 0 $	*D*4
$	4 ***num*** 6 * 7 *T* 5 + 1 *E* 0 $	*R*(*F* → ***num***)
$	8 *F* 6 * 7 *T* 5 + 1 *E* 0 $	*R*(*T* → *T* * *F*)
$	7 *T* 5 + 1 *E* 0 $	*R*(*E* → *E* + *T*)
$	1 *E* 0 $	ACP

Ambigüedad: Alteración de la gramática

else ambiguo (*dangling else*)

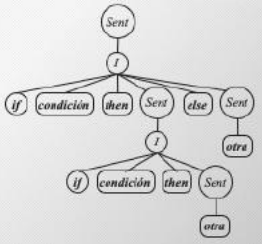
- La construcción habitual ***if-then-else*** de los lenguajes de programación puede reconocerse con las producciones:

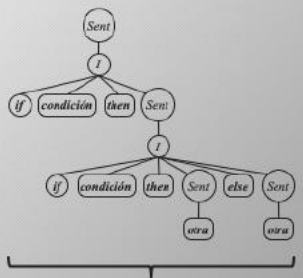
  $Sent \longrightarrow I \mid$ ***otra***
  $I \longrightarrow$ ***if condición then*** $Sent$
  $I \longrightarrow$ ***if condición then*** $Sent$ ***else*** $Sent$

- Esta gramática incurre en ambigüedades, la sentencia:

  if condición then if condición then otra else otra

- Tiene dos construcciones:





Construcción habitual en los lenguajes de programación, el ***else*** se empareja con el ***if*** más cercano.

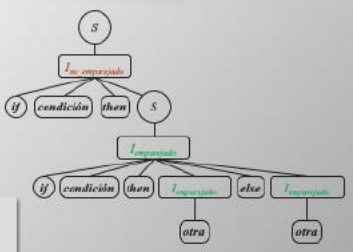
¿Cómo eliminar la ambigüedad y tener esta construcción?

Ambigüedad: Alteración de la gramática

else ambiguo (*dangling else*)

- Modificar la gramática:

$S \longrightarrow I \mid \textbf{otra}$
$I \longrightarrow \textbf{if condición then } S$
$I \longrightarrow \textbf{if condición then } S \textbf{ else } S$



if condición then if condición then otra else otra

$S \longrightarrow I_{emparejado} \mid I_{no\_emparejado}$
$I_{emparejado} \longrightarrow \textbf{if condición then } I_{emparejado} \textbf{ else } I_{emparejado} \mid \textbf{otra}$
$I_{no\_emparejado} \longrightarrow \textbf{if condición then } S \mid \textbf{if condición then } I_{emparejado} \textbf{ else } I_{no\_emparejado}$

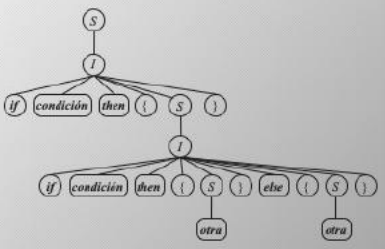
Ambigüedad: Alteración de la gramática

else ambiguo (*dangling else*)

- Modificar la gramática añadiendo delimitadores, Algol68 *if…fi*, Ada *if…end*, C {…}

$S \longrightarrow I \mid \textbf{otra}$
$I \longrightarrow \textbf{if condición then } S$
$I \longrightarrow \textbf{if condición then } S \textbf{ else } S$

if condición then { if condición then { otra } else { otra } }



$S \longrightarrow I \mid \textbf{otra}$
$I \longrightarrow \textbf{if condición then } \{\ S\ \}$
$I \longrightarrow \textbf{if condición then } \{\ S\ \} \textbf{ else } \{\ S\ \}$

Ambigüedad: Forma explícita

Gramática de operadores

- Para la gramática:
  1. $E \longrightarrow E + E$
  2. $E \longrightarrow E * E$
  3. $E \longrightarrow \textbf{num}$
- La tabla *SLR*(1) tiene conflictos *desplazamiento-reducción*

State	*	+	num	$	E
0			D2		1
1	D4	D3		ACP	
2	R3	R3		R3	
3			D2		5
4			D2		6
5	R1 | D4	R1 | D3		R1	
6	R2 | D4	R2 | D3		R2	

- Para resolver la ambigüedad de forma explícita hay que tomar la decisión de la acción a realizar
- Con la tabla es difícil tomar la decisión, es más fácil observando los ítems de los estados y conociendo el contenido de la pila para determinar si, para este caso, conviene *reducir* o *desplazar*

E_5: $E \longrightarrow E + E \cdot$ — *Acción*[5, {$, *, +}]= *R1*
$E \longrightarrow E \cdot + E$ — *Acción*[5, +]= *D3*
$E \longrightarrow E \cdot * E$ — *Acción*[5, *]= *D4*

E_6: $E \longrightarrow E * E \cdot$ — *Acción*[6, {$, *, +}]= *R2*
$E \longrightarrow E \cdot + E$ — *Acción*[6, +]= *D3*
$E \longrightarrow E \cdot * E$ — *Acción*[6, *]= *D4*

Ambigüedad: Forma explícita

Gramática de operadores

- El estado 5 se corresponde con una situación en la que la cima de la pila contiene los símbolos $E + E$ (por eso está lista para reducirse)

Entrada	Pila	Acción
$num + num\ X \ldots \$$	$0\ \$$	$D2$
$+\ num\ X \ldots \$$	$2\ num\ 0\ \$$	$R(E \rightarrow num)$
$+\ num\ X \ldots \$$	$1\ E\ 0\ \$$	$D3$
$num\ X \ldots \$$	$3 + 1\ E\ 0\ \$$	$D2$
$X \ldots \$$	$2\ num\ 3 + 1\ E\ 0\ \$$	$R(E \rightarrow num)$
$X \ldots \$$	$5\ E\ 3 + 1\ E\ 0\ \$$	???

- Si X es **$**, no hay problema, se reduce, si es ***num*** provocaría un error

- Si X es +, entonces aplicando la regla de asociatividad por la izquierda, debemos reducir, equivale a:

  $num + num + \ldots \Rightarrow (num + num) + \ldots$

- Si X es *, entonces aplicando la precedencia entre operadores, debemos desplazar, equivale a:

  $num + num * \ldots \Rightarrow num + (\ num\ * \ldots) \ldots$

State	*	+	num	$	E
0			D2		1
1	D4	D3		ACP	
2	R3	R3		R3	
3			D2		5
4			D2		6
5	R1 | D4	R1 | D3		R1	
6	R2 | D4	R2 | D3		R2	

State	*	+	num	$	E
0			D2		1
1	D4	D3		ACP	
2	R3	R3		R3	
3			D2		5
4			D2		6
5	D4	R1		R1	
6	R2 | D4	R2 | D3		R2	

Ambigüedad: Forma explícita

Gramática de operadores

- El estado 6 se corresponde con una situación en la que la cima de la pila contiene los símbolos $E * E$ (por eso está lista para reducirse)

Entrada	Pila	Acción
$num * num\ X \ldots \$$	$0\ \$$	$D2$
$* num\ X \ldots \$$	$2\ num\ 0\ \$$	$R(E \rightarrow num)$
$* num\ X \ldots \$$	$1\ E\ 0\ \$$	$D4$
$num\ X \ldots \$$	$4 * 1\ E\ 0\ \$$	$D2$
$X \ldots \$$	$2\ num\ 3 * 1\ E\ 0\ \$$	$R(E \rightarrow num)$
$X \ldots \$$	$6\ E\ 3 * 1\ E\ 0\ \$$	???

- Si X es $, no hay problema, se reduce, si es ***num*** provocaría un error

- Si X es +, entonces aplicando la regla de la precedencia entre operadores, debemos reducir, equivale a:

  $num * num + \ldots \Rightarrow (num * num) + \ldots$

- Si X es *, entonces aplicando la asociatividad por la izquierda, debemos reducir, equivale a:

  $num * num * \ldots \Rightarrow (num * num) * \ldots$

State	*	+	num	$	E
0			D2		1
1	D4	D3		ACP	
2	R3	R3		R3	
3			D2		5
4			D2		6
5	D4	R1		R1	
6	R2 | D4	R2 | D3		R2	

State	*	+	num	$	E
0			D2		1
1	D4	D3		ACP	
2	R3	R3		R3	
3			D2		5
4			D2		6
5	D4	R1		R1	
6	R2	R2		R2	

Ambigüedad: Forma explícita

else ambiguo (*dangling else*)

- Para la gramática:
  1. $S \rightarrow$ ***otra***
  2. $S \rightarrow I$
  3. $I \rightarrow$ ***if condición then S***
  4. $I \rightarrow$ ***if condición then S else S***
- La tabla *SLR*(1) tiene conflictos *desplazamiento-reducción*

State	condicion	else	if	otra	then	$	S	I
0			D4	D2			1	3
1						ACP		
2		R1				R1		
3		R2				R2		
4	D5							
5					D6			
6			D4	D2			7	3
7		R3 | D8				R3		
8			D4	D2			9	3
9		R4				R4		

- De nuevo, observando los ítems del estado 7, se puede tomar la decisión que permita asociar el ***else*** con el ***if*** más cercano

E_7: $I \rightarrow$ ***if condición then*** $S \cdot$
Acción[7, {**$**, ***else***}] = *R3*
$I \rightarrow$ ***if condición then*** $S \cdot$ ***else*** S
Acción[7, ***else***]= *D8*

Ambigüedad: Forma explícita

else ambiguo (*dangling else*)

- El estado 7 se corresponde con una situación en la que la cima de la pila contiene los símbolos ***if condición then otra*** (por eso está lista para reducirse)

Entrada	Pila	Acción
if condición then otra X … $	0 $	*D4*
condición then otra X … $	4 ***if*** 0 $	*D5*
then otra X … $	5 ***condición*** 4 ***if*** 0 $	*D6*
otra X … $	6 ***then*** 5 ***condición*** 4 ***if*** 0 $	*D2*
X … $	2 ***otra*** 6 ***then*** 5 ***condición*** 4 ***if*** 0 $	*R(S → otra)*
X … $	7 S 6 ***then*** 5 ***condición*** 4 ***if*** 0 $	???

- Si ***X*** es **$**, no hay problema, se reduce, si es ***condición if otra*** provocaría un error

- Si ***X*** es ***else*** entonces para concuerde con el ***if*** más próximo se debe desplazar

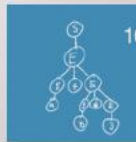
State	condicion	else	if	otra	then	$	S	I
0			D4	D2			1	3
1						ACP		
2		R1				R1		
3		R2				R2		
4	D5							
5					D6			
6			D4	D2			7	3
7		R3 | D8				R3		
8			D4	D2			9	3
9		R4				R4		

State	condicion	else	if	otra	then	$	S	I
0			D4	D2			1	3
1						ACP		
2		R1				R1		
3		R2				R2		
4	D5							
5					D6			
6			D4	D2			7	3
7		D8				R3		
8			D4	D2			9	3
9		R4				R4		

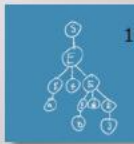
Ambigüedad

- No hay algoritmos que determinen si una gramática es ambigua
  - No hay algoritmos que conviertan una gramática ambigua en una equivalente no ambigua
- A veces es útil emplear una gramática ambigua
  - Construcciones más naturales y concisas
  - Aislar casos particulares que puede ser útil tenerlos separados
- Las gramáticas ambiguas sólo deben usarse de forma escasa y controlada, para asegurar qué lenguaje se reconoce

Tratamiento de errores



Rutinas de Manejo de Errores



- Ocupan gran parte de los compiladores
- Características
  - Informar con claridad y exactitud (tipo de error, línea en que se produce,…)
  - Recuperación rápida para seguir con el análisis
  - No debe retrasar el procesamiento de programas sin errores
- Acciones posibles
  - Detectar errores
  - Informar los errores
  - Recuperar de los errores

Tipos de Errores

- Tipos de errores
  - Léxicos: escribir mal un identificador
  - Sintácticos: no poner un ";" al final de una sentencia
  - Semánticos: multiplicar una variable booleana
  - Lógicos: bucle infinito
- Se minimiza el número de errores...
  - Léxicos
    - Si se utiliza alguna herramienta que complete palabras
  - Sintácticos
    - Si se utiliza algún editor basado en sintaxis

Objetivos

- Objetivos
  - Detectar rápidamente el error
  - Detectar todos los errores
  - Evitar detección incorrecta de errores falsos
  - Evitar errores repetidos
- Ejemplo

```
void main (){
int i, j, k;
i=0 ; /*asigno 0 a i /
j=;
=k;
}
```

Objetivos

- Evitar detección incorrecta de errores falsos

```
void main (){          //linea 1
int i, j;              //linea 2
i=1;                   //linea 3
while (i) {            //linea 4
   int j=i+1;          //linea 5
   i=i-1;              //linea 6
   }                   //linea 7
j=2;                   //linea 8
if (i<j) j++;          //linea 9
j=i-1;                 //linea 10
}                      //linea 11
ERRORES:
```

Objetivos

- Evitar errores repetidos

```
void main (){          //linea 1
int i, j;              //linea 2
int a,n=10;            //linea 3
i=1;                   //linea 4
for (j=0;j<n;j++)      //linea 5
   a=j-i+1;            //linea 6
   b=2*a+j;            //linea 7
   }                   //linea 8
…
}
ERRORES:
```

Gestión de errores

- El analizador debe ser capaz de reconocer errores y emitir mensajes de información
- Gestión ideal:
  - I. Identificación de error
    - Temprano
  - II. Localización
    - Fila y columna
  - III. Emisión de mensaje
    - Específicos
  - IV. Recuperación de error
    - Continuar lo más posible

Estrategias de Recuperación

- No hay una estrategia de aceptación universal
- Las principales estrategias de recuperación son:
  - Modo de pánico
  - Nivel de frase
  - Producciones de error
  - Corrección Global

Modo de Pánico

- Características
  - Método más sencillo
  - Lo pueden usar la mayoría de los AS
  - No entra en lazos infinitos
  - Adecuado para lenguajes en los que son raros múltiples errores en la misma proposición
- Funcionamiento
  - El AS desecha símbolos de la entrada, uno por uno
  - Hasta encontrar un token de sincronización
    - Delimitadores (punto y coma, palabras clave como end)
- Inconvenientes
  - Se omiten gran cantidad de símbolos sin analizar

A nivel de frase

- Características
  - Pueden corregir cualquier cadena de entrada
- Funcionamiento
  - Descubierto el error se corrige (localmente) la entrada por un prefijo que permite continuar el AS
    - Sustituir una coma por un punto y coma, insertar un punto y coma, etc.
- Inconvenientes
  - Dificultad para resolver situaciones en las que el error se produjo antes de la detección de este (por ejemplo: se olvidó poner un punto y coma … ¿Dónde lo ponemos?)
  - Pueden producir lazos infinitos
    - Evitar insertar símbolos antes del símbolo actual en la entrada

Producciones de error

- Funcionamiento
  - Conocidos los errores más comunes, se extiende la gramática con producciones de error
  - Reconocido el error, se dan diagnósticos precisos de la construcción errónea

Ejemplo: supongamos que en nuestro lenguaje se realizan Asignaciones por medio del token ':= ' y compareciones por '=='

S ⟶ id := num | id == num (asig. Y comp.)

S ⟶ id = num {producción añadida de ERROR, la gramática ahora reconoce id = num, pero sabemos que esa construcción es errónea: Podemos continuar con el análisis pero no generaremos código}

Corrección Global

- Características
  - Algoritmos que eligen una secuencia mínima de cambios para obtener una corrección global de menor costo
  - Ej.: `x=a(p+q(-b(r-s);` ⟶ `a(p+q)-b(r-s);`
    `ifa=b then sum=0;` ⟶ `if a =b then sum=0;`
- Inconvenientes
  - Técnicas costosas en tiempo y espacio: métricas de distancias, búsqueda, optimización, …

Recuperación Analizador Descendente

- Gramática:

  S → a A S | b A
  A → c A | d

- Tabla:

Σ_N	a	b	c	d
S	aAS	bA		
A			cA	d

- Entrada: a b ...
- Estado del análisis cuando detecta el error:

  Pila Entrada
  $S a b ...
  $SAa a b ...
  $SA b ...

- Error: Se ha encontrado una b, cuando se esperaba una c o d

Recuperación en AS LL(1)

- Detección del error
  - El terminal de la cima de la pila no concuerda con la entrada
  - El no terminal A de la cima de la pila y el termina a de la entrada indexan la tabla de análisis sintáctico una entrada vacía, `M[A,a]=∅`
- Recuperación
  - Modo pánico
    - La eficacia de esta estrategia depende de la elección del conjunto de tokens de sincronización
    - La elección debe ser tal que el AS se recupere rápidamente de los errores más comunes

Recuperación en AS LL(1)

- Dado un no terminal A en la cima de la pila y un token que no tiene acción en asociada (no está en *Primero(A)*, ni *Siguiente(A)* si $\lambda \in Primero(A)$):
- Alternativas
  - Extraer A de la pila
  - Extraer tokens de la entrada hasta poder continuar el análisis sintáctico
  - Insertar un nuevo no terminal en la pila
- Estrategia según heurísticas

Recuperación en AS LL(1)

- Modo pánico. Heurísticas
  - Colocar todos los símbolos de *SIGUIENTE(A)* en el conjunto de sincronización de *A*
    - Saltando tokens hasta encontrar uno que pertenezca a *SIGUIENTE(A)* y sacar *A* de la pila, probablemente el AS siga
    - Acción especial si se vacía la pila: poner *S* y saltar hasta *PRIMERO(S)*
  - Añadir al conjunto de sincronización tokens de inicio de bloques de una jerarquía superior
    - Ej.: al extender el conjunto *SIGUIENTE* se puede producir que al faltar un ';', el AS salte a un inicio de bloque que le sigue
  - Añadir los símbolos de *PRIMERO(A)* al conjunto de sincronización de *A*
  - Si $\lambda \in PRIMERO(A)$ se puede usar por omisión la producción $A \rightarrow \lambda$
    - Se puede posponer la detección del error pero no su omisión
  - Si la cima de la pila no coincide con la entrada, se inserta en la pila y se lee de la entrada informando del proceso

Recuperación Analizador LL(1)

- Modo Pánico
- Ejemplo:

```
E  → TE'
E' → +TE' | λ
T  → FT'
T' → *FT' | λ
F  → (E) | id
```

Entrada:

```
)id*+id
```

Σ_N	Primero	Siguiente
E	{(,id}	{),$}
E'	{+,λ}	{),$}
T	{(,id}	{+,),$}
T'	{*,λ}	{+,),$}
F	{(,id}	{+,*),$}

	id	+	*	(	)	$
E	E→TE'			E→TE'	sinc	sinc
E'		E'→+TE'			E'→λ	E'→λ
T	T→FT'	sinc		T→FT'	sinc	sinc
T'		T'→λ	T'→*FT'		T'→λ	T'→λ
F	F→id	sinc	sinc	F→(E)	sinc	sinc

Recuperación Analizador LL(1)

- A nivel de frase
  - Se llenan las entradas en blanco de la tabla con llamadas a rutinas de error
    - Las rutinas insertan, cambian o eliminan símbolos de la entrada y emiten mensajes de error
    - Pueden sacar elementos de la pila
  - Hay que asegurarse de que no se produzcan lazos infinitos
    - Las acciones de recuperación deben consumir símbolos de la entrada o de la pila

Errores en Análisis Ascedente

- Gramática

S ⟶ A A
A ⟶ x A | y

	acción			ir_a	
	x	y	$	S	A
0	d3	d4	e	1	2
1	e	e	acpt		
2	d3	d4	e	5	
3	d3	d4	e	6	6
4	r3	r3	r3		7
5	r1	e	e		8
6	r2	r2	r2		

'x' inesperado

'y' inesperado

Fin inesperado

Errores en Análisis Ascedente

- Gramática

S ⟶ A A
A ⟶ x A | y

- Entrada: xy
- Estado del análisis cuando detecta el error:

Pila	Entrada
0x3A6	$
0A2	$
0A2	$

- Se podría añadir y

	acción			ir_a	
	x	y	$	S	A
0	d3	d4		1	2
1			acpt		
2	d3	d4	**err3**	5	
3	d3	d4		6	6
4	r3	r3	r3		7
5	r1				8
6	r2	r2	r2		

Detección con AS LR(1)

- Detección del error
  - Al consultar la tabla de acciones del AS y la entrada es de error
    - Detección más rápida que otros analizadores: propiedad del prefijo viable
  - Nunca se hace un desplazamiento posterior al token erróneao
    - LR canónico nunca hace una reducción antes de anunciar un error
    - SLR y LALR pueden realizar varias reducciones antes de anunciar un error, pero nunca desplazan un símbolo de la entrada erróneo
    - Mayor rapidez de detección al aumentar estados

Detección errores AS LR(1)

- Ej. LALR vs LR(1). Gramática: A⟶ (A) | a

E	acción				ir_a
	(	a	)	$	A
0	d3	d2			1
1				acpt	
2			r2	r2	
3	d3	d2			4
4			d5		
5			r1	r1	

E	acción				ir_a
	(	a	)	$	A
0	d2	d3			1
1				acpt	
2	d5	d6			4
3				r2	
4			d7		
5	d5	d6			8
6			r2		
7				r1	
8			d9		
9			r1		

- Comparar: (a$
  a)$

Recuperación AS LR(1)

- Heurística recuperación con LR(1) - modo pánico
  - Se examina la pila hasta encontrar un estado `s` con valor de `ir_a` para un un símbolo no terminal `A`
  - Se desechan símbolos de la entrada hasta encontrar un terminal a válido para el no terminal `A`. Si hay acción sobre el token actual en uno de los estados en `ir_a`, insertarlo. Si hay varios, preferir desplazamiento a reducción.
  - Poner en la pila el estado de `ir_a[s,A]` y sigue el análisis sintáctico
  - Si no hay acción, avanzar entrada

Recuperación con AS LR(1)

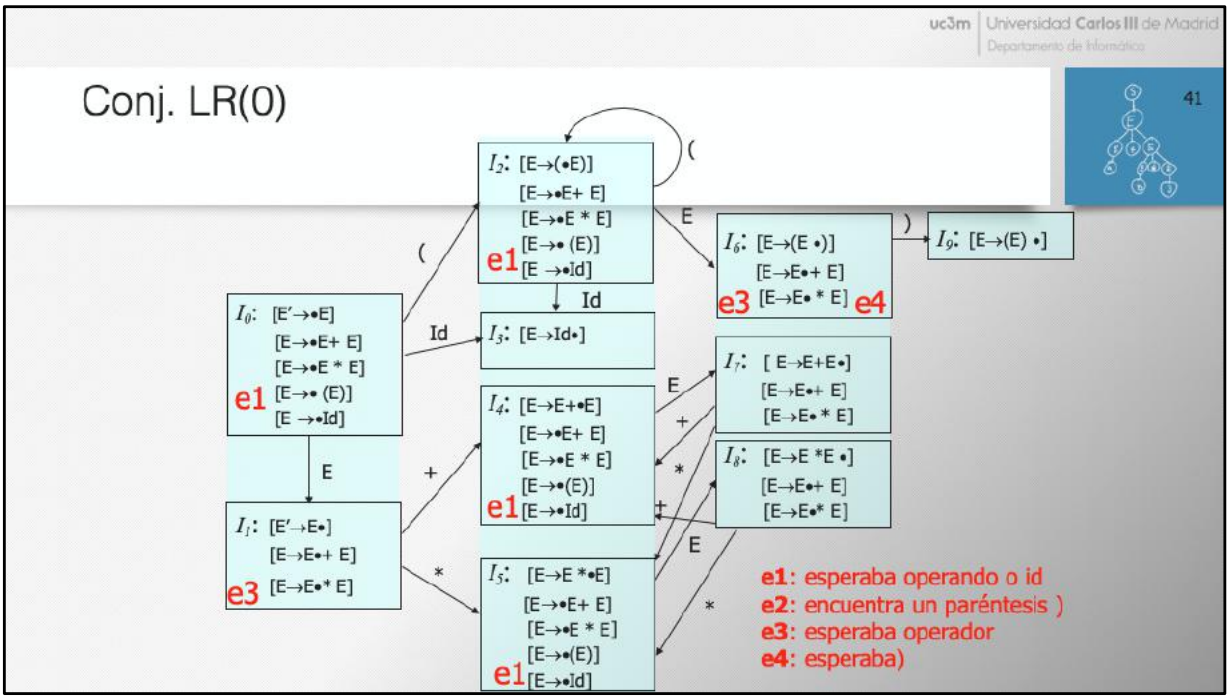
- A nivel de frase
  - Procedimientos que modifican la pila y/o la entrada dependiendo del error
  - Recuperación: eliminarán o desplazarán al menos un símbolo de la entrada o reducirán la pila
  - Evitar extraer de la pila estados que alcancen un no terminal (construcciones correctamente examinadas)
  - Evitar excesivos mensajes de error: cuando las reducciones son al mismo estado, extender la reducción a entradas en blanco (los errores se detectarán ahora al desplazar)

Ejemplo recuperación LR

- A nivel de frase
- Ejemplo: `id+)`
- `E → E+E|E*E|(E)|id`

	acción						ir_a
	id	+	*	(	)	$	E
0	d3			d2			1
1		d4	d5			acpt	
2	d3			d2		e1	6
3		r4	r4		r4	r4	
4	d3			d2			7
5	d3			d2			8
6		d4	d5		d9		
7		r1	d5		r1	r1	
8		r2	r2		r2	r2	
9		r3	r3		r3	r3	

Conj. LR(0)


I0: [E'→•E]
[E→•E+ E]
[E→•E * E]
[E→• (E)]
[E →•Id]
e1
I1: [E'→E•]
[E→E•+ E]
e3 [E→E•* E]
I2: [E→(•E)]
[E→•E+ E]
[E→•E * E]
[E→• (E)]
e1 [E →•Id]
I3: [E→Id•]
I4: [E→E+•E]
[E→•E+ E]
[E→•E * E]
[E→•(E)]
e1 [E→•Id]
I5: [E→E *•E]
[E→•E+ E]
[E→•E * E]
[E→•(E)]
e1 [E→•Id]
I6: [E→(E •)]
[E→E•+ E]
e3 [E→E• * E] e4
I7: [E→E+E•]
[E→E•+ E]
[E→E• * E]
I8: [E→E *E •]
[E→E•+ E]
[E→E•* E]
I9: [E→(E) •]
(
Id
E
+
*
)
e1: esperaba operando o id
e2: encuentra un paréntesis)
e3: esperaba operador
e4: esperaba)

Recuperación AS LR(1)

- A nivel de frase
  - Ejemplo: `id+)`

  `E → E+E|E*E|(E)|id`

- e1: se esperaba un operando `id` ó `(` pero se encontró operador o final
  - introducir `id` en la pila y estado 3
  - mostrar "falta operando"
- e2: encuentra un paréntesis
  - eliminar `)` de la entrada
  - mostrar "`)` no equilibrado"
- e3: se esperaba un operador pero se encontró `id` ó `)`
  - introducir `+` en la pila y estado 4
  - mostrar "falta operador"
- e4: se esperaba operador ó `)` pero encontró el final de la entrada
  - introducir `)` en la pila y estado 9
  - mostrar "falta paréntesis derecho"

	acción						ir_a
	id	+	*	(	)	$	E
0	d3	e1	e1	d2	e2	e1	1
1	e3	d4	d5	e3	e2	acpt	
2	d3	e1	e1	d2	e2	e1	6
3	r4	r4	r4	r4	r4	r4	
4	d3	e1	e1	d2	e2	e1	7
5	d3	e1	e1	d2	e2	e1	8
6	e3	d4	d5	e3	d9	e4	
7	r1	r1	d5	r1	r1	r1	
8	r2	r2	r2	r2	r2	r2	
9	r3	r3	r3	r3	r3	r3	

Errores con yacc/cup

- yacc/cup incorporan la detección de errores
  - Los mensajes de error son mas informados
- La función que informa de los errores tiene que implementarse

```
int yyerror (char *s) {fprintf (stderr, "%s ",s)};
```

- Es conveniente ampliarla para mostrar el número de línea, el lexema de error, etc.

Errores con yacc/cup

- El token **error** permite recuperarse de los errores mejorando el informe del error

```
E: IF ´(´ cond ´)´
| IF ´(´ error ´)´ {yyerror("##Falta condición");}
```

- Es una recuperación en modo pánico, contextualizada en una producción
  - Búsqueda de un estado concreto de la pila para comenzar la búsqueda del token de sincronización

Errores con yacc/cup

- Comportamiento general de recuperación de errores
  1. Ante situación de error (no acción), extrae de la pila hasta localizar estado con el item error como búsqueda legal
  2. Desplaza el token error
  3. A continuación elimina tokens de la entrada hasta que puede "reducir" la producción del error
     - Supone una recuperación para continuar procesando

```
E: IF ´(´ cond ´)´
| IF ´(´ error ´)´
```

Busca estado con item
E→IF (• error)

Uso de "token" error

```
%token NUMERO, MAS, POR, PAR_I, PAR_D
%start expr /* simbolo axioma sentencial */

%%
expr:  expr MAS term
    |term
;
term:  term POR factor
    |factor
;
factor: NUMERO
        |PAR_I expr error {yyerror("##ERROR parentesis\n");}
    |error
;

%%
. . .
```

Conclusiones de Manejo de Errores

- Aspecto complejo del diseño del compilador
  - Análisis cuidadoso del lenguaje y posibles errores
  - No hay una estrategia de aceptación universal
  - Abundan técnicas heurísticas y ad hoc
- Objetivos
  - Informar con claridad, exactitud y extensión
  - Evitar errores en cascada y procesos infinitos
- Principio general de recuperación
  - Minimizar tokens eliminados/modificados
  - Dejar el analizador listo para continuar procesando
- Analizadores LL y LR gran capacidad para estrategias en modo pánico y frase en función del contexto
- Necesidad de recuperación?