
Análisis Sintáctico (II)

Diseño de Lenguajes de Programación

Ingeniería Informática

Universidad de Oviedo

(v1.12)

Raúl Izquierdo Castanedo

En capítulos anteriores...

Estrategia ascendente

- Desde la cadena hasta el símbolo inicial

Técnica de Reducción por Desplazamiento

- En cada paso asegura pivote y/o regla a elegir
- Basado en una pila
 - Comienza vacía
 - Operaciones
 - Shift: Introducir token
 - Reduce. Dada $n \rightarrow \beta$ extrae β **de la cima** e introduce n
 - Acaba con s si pertenece al lenguaje

Autómata Generado por Yacc

Análisis Sintáctico

Autómata (I)

Yacc genera un autómata

- Lo recorre en tiempo de ejecución
 - Nos lo describe en un fichero de texto

Es necesario saber interpretar el autómata para:

1. Si la gramática no es LR(1), hay que saber qué parte modificar
2. Activar modo de traza

Yacc es más cómodo que hacerlo a mano

- Si se sabe utilizar cuando las cosas van mal

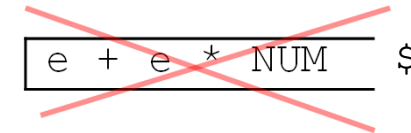
Autómata LR(1)

¿Qué representa cada estado de este autómata?

- Una posible *situación de la pila* (una combinación de símbolos en la misma)
 - Y las acciones que hay que hacer en esa situación (shift o reduce) en función del siguiente token

El algoritmo de reconocimiento LR(1) (implementado en las versiones de Yacc)

- Calcula todas las situaciones posibles de la pila (estados) y decide de antemano qué habrá que hacer en cada una
 - Más rápido que buscar pivotes con cada shift *en tiempo de ejecución*



Nuestro objetivo

- No es saber construir el autómata ("Compiler Design in C". Allen Holub)

Ejemplo

s: NUM IDENT (regla 1)

- Para la regla anterior habría tres estados (simplificando)

Estado 0

s: . NUM IDENT

- shift NUM y pasar a 1

Estado 1

s: NUM . IDENT

- shift IDENT y pasar a 2

Estado 2

s: NUM IDENT .

- Reduce por regla 1

Autómata

Cómo obtener el autómata

s: DATA a b

a: IDENT

b: IDENT
| IDENT NUM

Opción -v

c:\>yacc -v ejemplo.y



*Se genera el fichero
y.output*

y.output

state 0

\$accept : . s \$end (0)

"DATA" shift 1

. error

s goto 2

state 1

s : "DATA" . a b (1)

"IDENT" shift 3

. error

a goto 4

state 2

\$accept : s . \$end (0)

\$end accept

state 3

a : "IDENT" . (2)

. reduce 2

state 4

s : "DATA" a . b (1)

"IDENT" shift 5

. error

b goto 6

state 5

b : "IDENT" . (3)

b : "IDENT" . "NUM" (4)

"NUM" shift 7

\$end reduce 3

state 6

s : "DATA" a b . (1)

. reduce 1

state 7

b : "IDENT" "NUM" . (4)

. reduce 4

Algoritmo de Reconocimiento

```
Repetir {  
  si acción[estado, token] = shift  $e_n$   
    Meter token y  $e_n$   
  else si acción[estado, token] = reduce por  $n \rightarrow \beta$   
    Sacar  $|\beta|$  símbolos y  $|\beta|$  estados de la pila  
    Sea e el estado en la cima de la pila  
    Meter n  
    Meter estado Goto[e, n]  
  else si acción[estado, token] = aceptar  
    return Pertenece;  
  else  
    return NoPertenece;  
}
```

Autómata LR(1)

y.output

	0	DATA x y 80 \$
shift	0 DATA 1	x y 80 \$
shift	0 DATA 1 ID 3	y 80 \$
a: ID	0 DATA 1 a	y 80 \$
	0 DATA 1 a 4	y 80 \$
shift	0 DATA 1 a 4 ID 5	80 \$
shift	0 DATA 1 a 4 ID 5 NUM 7	\$
b: ID NUM	0 DATA 1 a 4 b	\$
	0 DATA 1 a 4 b 6	\$
s: DATA a b	0 s	\$
	0 s 2	\$

state 0

\$accept : . s \$end (0)

"DATA" shift 1

. error

s goto 2

state 1

s : "DATA" . a b (1)

"IDENT" shift 3

. error

a goto 4

state 2

\$accept : s . \$end (0)

\$end accept

state 3

a : "IDENT" . (2)

. reduce 2

state 4

s : "DATA" a . b (1)

"IDENT" shift 5

. error

b goto 6

state 5

b : "IDENT" . (3)

b : "IDENT" . "NUM" (4)

"NUM" shift 7

\$end reduce 3

state 6

s : "DATA" a b . (1)

. reduce 1

state 7

b : "IDENT" "NUM" . (4)

. reduce 4

No indica el estado al que ir!!!

Autómata LR(1)

	Acción					Goto		
	DATA	ID	NUM	\$		s	a	b
0	s1					2		
1		s3					4	
2				ac				
3	r2	r2	r2	r2				
4		s5						6
5			s7	r3				
6	r1	r1	r1	r1				
7	r4	r4	r4	r4				

s: DATA a b (1)
 a: IDENT (2)
 b: IDENT (3)
 | IDENT NUM (4)

y.output

state 0

\$accept: . s \$end (0)

"DATA" shift 1

. error

s goto 2

state 1

s : "DATA" . a b (1)

"IDENT" shift 3

. error

a goto 4

state 2

\$accept : s . \$end (0)

\$end accept

state 3

a : "IDENT" . (2)

. reduce 2

state 4

s : "DATA" a . b (1)

"IDENT" shift 5

. error

b goto 6

state 5

b : "IDENT" . (3)

b : "IDENT" . "NUM" (4)

"NUM" shift 7

\$end reduce 3

state 6

s : "DATA" a b . (1)

. reduce 1

state 7

b : "IDENT" "NUM" . (4)

. reduce 4

Ejercicio E1

Gramática

%%

s: exp

exp: exp '+' factor
| factor

factor: NUM

%%

A partir del fichero "y.output":

1. Hacer una traza de la entrada
30 + 40
2. Hacer la tabla del autómata LR(1)

y.output

state 0

\$accept : . s \$end (0)

"NUM" shift 1

. error

s goto 2

exp goto 3

factor goto 4

state 1

factor : "NUM" . (4)

. reduce 4

state 2

\$accept : s . \$end (0)

\$end accept

state 3

s : exp . (1)

exp : exp . '+' factor (2)

'+' shift 5

\$end reduce 1

state 4

exp : factor . (3)

. reduce 3

state 5

exp : exp '+' . factor (2)

"NUM" shift 1

. error

factor goto 6

state 6

exp : exp '+' factor . (2)

. reduce 2

Traza de Ejecución

Cómo se activa

```
public static void main(String[] args)
{
    ...
    Parser parser=new Parser(lex, true);
    ...
}
```

Ejemplo anterior

	0	DATA x y 80 \$
shift	0 DATA 1	x y 80 \$
shift	0 DATA 1 ID 3	y 80 \$
a: ID	0 DATA 1 a	y 80 \$
	0 DATA 1 a 4	y 80 \$
shift	0 DATA 1 a 4 ID 5	80 \$
shift	0 DATA 1 a 4 ID 5 NUM 7	\$
b: ID NUM	0 DATA 1 a 4 b	\$
	0 DATA 1 a 4 b 6	\$
s:DATA a b	0 s	\$
	0 s 2	\$

Salida estándar

```
loop
yyn:0 state:0 yychar:-1
next yychar:257 (257 = DATA)
state 0, shifting to state 1
```

```
loop
yyn:0 state:1 yychar:-1
next yychar:258 (258 = ID)
state 1, shifting to state 3
```

```
loop
state 3, reducing 1 by rule 2 (a : "IDENT")
reduce
after reduction, shifting from state 1 to state 4
```

```
loop
yyn:0 state:4 yychar:-1
next yychar:258 (258 = ID)
state 4, shifting to state 5
```

```
loop
yyn:0 state:5 yychar:-1
next yychar:259 (259 = NUM)
state 5, shifting to state 7
```

```
loop
state 7, reducing 2 by rule 4 (b : "IDENT" "NUM")
reduce
after reduction, shifting from state 4 to state 6
```

Utilidad de la Traza de Ejecución

Uso de la traza

- Dada una entrada, saber el por qué la rechaza

Supóngase la gramática:

```
s: DATA a b
a: IDENT
b: IDENT
  | IDENT NUM
```

Y la entrada:

DATA x 80

El analizador la rechaza
¿Por qué?

Traza de 'DATA x 80' (salida estándar)

```
loop
 yyn:0  state:0  yychar:-1
 next yychar:257      (257 = DATA)
 state 0, shifting to state 1

 yyn:0  state:1  yychar:-1
 next yychar:258      (258 = ID)
 state 1, shifting to state 3

 state 3, reducing 1 by rule 2 (a : "IDENT")
 reduce
 after reduction, shifting from state 1 to state 4

 yyn:0  state:4  yychar:-1
 next yychar:259      (259 = NUM)

error recovery discarding state 4
error recovery discarding state 1
error recovery discarding state 0
Error sintáctico: stack underflow. aborting...
```

Fragmento de *y.output* →

```
state 4
 s : "DATA" a . b (1)

"IDENT" shift 5
. error

b goto 6
```

Tratamiento de Conflictos s/r

Análisis Sintáctico

Conflictos s/r

En prácticas...

%left '+'
%left '*' } 

%0%

programa: sentencia
;

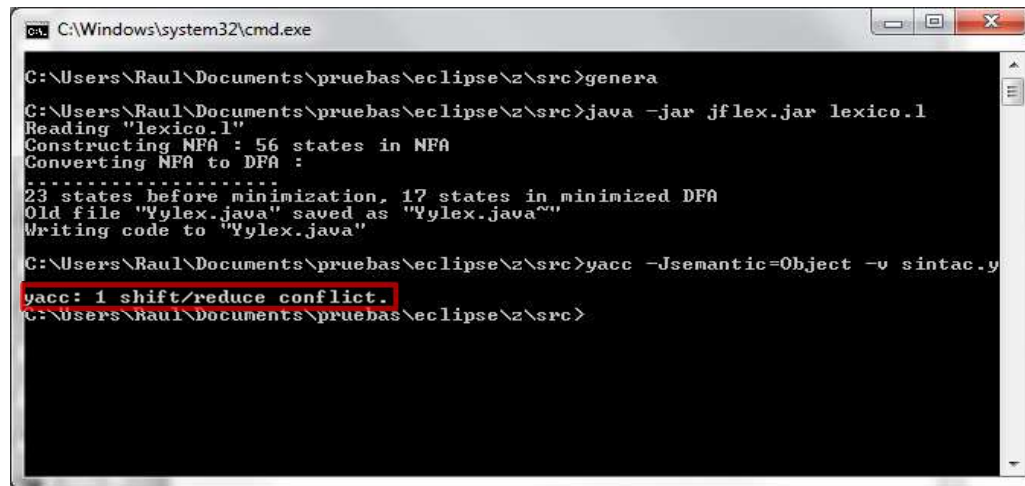
sentencia: 'PRINT' expr ';' ;

expr: 'LITENT'
| expr '+' expr
| expr '*' expr
;

%0%

Autómata LR(1)

Para que no salga esto...



```
C:\Windows\system32\cmd.exe

C:\Users\Raul\Documents\pruebas\eclipse\z\src>genera
C:\Users\Raul\Documents\pruebas\eclipse\z\src>java -jar jflex.jar lexico.l
Reading "lexico.l"
Constructing NFA : 56 states in NFA
Converting NFA to DFA :
.....
23 states before minimization, 17 states in minimized DFA
Old file "Yylex.java" saved as "Yylex.java~"
Writing code to "Yylex.java"

C:\Users\Raul\Documents\pruebas\eclipse\z\src>yacc -Jsemantic=Object -v sintac.y
yacc: 1 shift/reduce conflict.
C:\Users\Raul\Documents\pruebas\eclipse\z\src>
```

- ❑ ¿Qué quiere decir eso?
- ❑ ¿Por qué hay un conflicto en este caso?

En general, ¿Qué hacer cuando Yacc rechaza una gramática?

- Opciones
 1. Cambiar el Lenguaje
 2. Transformar Gramática
 3. Reglas de Selección

Estas son los
%left

1. Cambiar el lenguaje (I)

Ejemplo

%%

sent: PRINT

| IF cond THEN sent

| IF cond THEN sent ELSE sent

cond: TRUE | FALSE

%%

Informe de Yacc

- Conflicto shift/reduce
 - ¿Por qué?

IF false THEN IF false THEN print a ELSE print b



¿Con qué cambio del lenguaje se podría evitar esto?

1. Cambiar el lenguaje (II)

Se cambia esta gramática:

```
sent: PRINT
    | IF cond THEN sent
    | IF cond THEN sent ELSE sent
```

Por esta:

```
sent: PRINT
    | IF cond THEN '{' sent '}'
    | IF cond THEN '{' sent '}' ELSE '{' sent '}'
```

Ahora el lenguaje sería:

```
IF false THEN { IF false THEN { print a } } ELSE { print b }
```

```
IF false THEN { IF false THEN { print a } ELSE { print b } }
```

2. Transformar la Gramática

Cuando no se quiere cambiar el lenguaje

- Hallar gramática equivalente

Transformación más común

- Expansión de un no terminal

```
s: a X Y
   | b X Z
a: W
b: W
```

Informe de Yacc

```
1: reduce/reduce conflict (reduce 3, reduce 4) on 'X'
state 1
  a : 'W' . (3)
  b : 'W' . (4)
```

Expansión

```
s: W X Y
   | W X Z
```

En general, las reglas desglosadas suelen resultarle de más ayuda a yacc

3. Reglas de Selección (I)

No cambian la gramática

- Determinan qué árbol generar

En Ascendente

- Ante un mismo estado de la pila se puede hacer tanto un shift como un reduce
- Determinar acción 'a mano'
 - Determinar acción ante un token y un estado
 - Shift o reduce

3. Reglas de Selección (II)

Ejemplo

```
%%  
sent: PRINT  
    | IF cond THEN sent  
    | IF cond THEN sent ELSE sent  
    ;  
cond: TRUE | FALSE  
%%
```

Informe de Yacc

8: *shift/reduce conflict (shift 9, reduce 2) on "ELSE"*

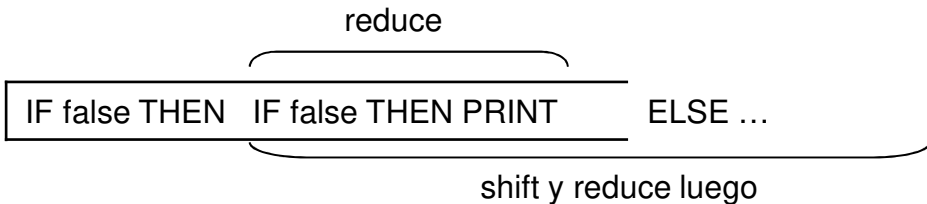
state 8

```
sent : "IF" cond "THEN" sent . (2)  
sent : "IF" cond "THEN" sent . "ELSE" sent (3)  
"ELSE" shift 9  
$end reduce 2
```

state 9

```
sent : "IF" cond "THEN" sent "ELSE" . sent (3)  
"PRINT" shift 1  
"IF" shift 2  
. error  
sent goto 10
```

- ¿Qué se quiere? ¿Cómo se le dice?



Prioridades en Yacc

Si yacc encuentra s/r comprueba si hay Regla de Selección

- Comprueba las prioridades del token y de la regla
 1. Si *alguno no tiene*, señala el conflicto.
 2. Si *ambos tienen* prioridad:
 - Si (*prioridad regla > prioridad token*), se *reduce*.
 - Si (*prioridad token > prioridad regla*), se hace *shift*.
 - Si (*prioridad regla == prioridad token*), se usa la asociatividad del token
$$3 - 2 - 1$$
 - Si es asociativo a izquierda: Reduce
 - ¿Por qué?
 - Si es asociativo a derecha: Shift
 - Si no es asociativo: conflicto

¿Cómo se indican las prioridades y la asociatividad?

Prioridades en Yacc

La prioridad de un terminal depende de su definición

- `%token` o comillas **no asignan** prioridad
- `%right`, `%left` y `%nonassoc` **sí la asignan**
 - Mayor prioridad cuanto *más abajo* se definan
 - Se diferencian en la asociatividad que asignan

```
%left W
%left X
%%
```

↓ Mayor prioridad

Prioridad de una regla

- La que se defina con `%prec` -----> c: h "Y" k %prec W;
- Si no `%prec` toma la de su último token -----> a: a "Y" X b

- Por tanto: una regla se queda **sin prioridad**...
 - ...si no tiene `%prec` y además
 - no hay terminales en la regla -----> b: c d
 - o el último no tiene prioridad (definido con `%token` o comillas) -----> c: 'X' 'K' h

3. Reglas de Selección (III)

Volvemos a...

%%

```
sent: PRINT
    | IF cond THEN sent
    | IF cond THEN sent ELSE sent
```

```
cond: TRUE | FALSE
```

%%

Informe de Yacc

8: shift/reduce conflict (shift 9, reduce 2) on "ELSE"

state 8

```
sent : "IF" cond "THEN" sent . (2)
sent : "IF" cond "THEN" sent . "ELSE" sent (3)
```

```
"ELSE" shift 9
$end reduce 2
```

¿Solución?

3. Reglas de Selección (IV)

Gramática 1

```
%nonassoc MENORQUEELSE
```

```
%nonassoc ELSE
```

```
%%
```

```
sent : PRINT
```

```
    | IF cond THEN sent                %prec MENORQUEELSE
```

```
    | IF cond THEN sent ELSE sent
```

Gramática 2

```
%nonassoc THEN
```

```
%nonassoc ELSE
```

```
%%
```

```
sent : PRINT
```

```
    | IF cond THEN sent    /* sin %prec */
```

```
    | IF cond THEN sent ELSE sent
```


Ejemplo (I)

%%

```
e: e '+' e
   | LITENT
```

%%

Informe de Yacc

4: shift/reduce conflict (shift 3, reduce 1) on '+'
state 4

e : e . '+' e (1)

e : e '+' e . (1)

'+' shift 3

\$end reduce 1

- ¿Cuál es el problema?
 - ¿Qué se quiere hacer?

Ejemplo (II)

Solución

```
%left '+'
```

```
%%
```

```
e: e '+' e
```

```
  | LITENT
```

```
%%
```

- ¿Qué prioridad tiene ahora cada elemento afectado?

Ejercicio E2

```
%left '+'
%%
expr: expr '+' expr
    | LITENT
    | '(' expr ')'
    | '(' TYPE ')' expr
```

```
%%
```

Informe de Yacc

```
10: shift/reduce conflict (shift 6, reduce 4) on '+'
```

```
state 10
```

```
expr : expr . '+' expr (1)
```

```
expr : '(' "TYPE" ')' expr . (4)
```

```
'+' shift 6
```

```
$end reduce 4
```

```
'(' reduce 4
```

- Describir la causa del conflicto mediante sentencia de ejemplo
- Plantear una solución con *cada alternativa de tratamiento* (Reglas de Selección, cambio lenguaje y transformación de gramática)

Otros Aspectos

Uso adecuado

- No hay que dar prioridad a todo token y regla
- Debe ser el último recurso
 - If/Else
 - Expresiones
 - Y ***muy*** poco mas...
- **En resumen, toda esta clase se podría resumir en...**
 - ...cuando añadáis un operador a la gramática, acordaos de declararlo con *%left*
 - Si no, yacc os lo recordará...

Tarea

Sea la gramática

```
%%  
e: e '+' e  
   | e '*' e  
   | LITENT  
%%
```

Informe de Yacc: 4 conflictos s/r

5: shift/reduce conflict (shift 3, reduce 1) on '+'

5: shift/reduce conflict (shift 4, reduce 1) on '*'

state 5

e : e . '+' e (1)

e : e '+' e . (1)

e : e . '*' e (2)

6: shift/reduce conflict (shift 3, reduce 2) on '+'

6: shift/reduce conflict (shift 4, reduce 2) on '*'

state 6

e : e . '+' e (1)

e : e . '*' e (2)

e : e '*' e . (2)

¿Solución válida?

```
%left '+' '*'  
%%
```

Soluciones

Análisis Sintáctico

Solución E1 (I)

	0	30 + 40 \$	state 0 \$accept : . s \$end (0)	state 4 exp : factor . (3)
shift	0 NUM 1	+ 40 \$	"NUM" shift 1	. reduce 3
factor: NUM	0 factor	+ 40 \$. error	state 5 exp : exp '+' . factor (2)
	0 factor 4	+ 40 \$	s goto 2	
exp: factor	0 e	+ 40 \$	exp goto 3	"NUM" shift 1
	0 e 3	+ 40 \$	factor goto 4	
shift	0 e 3 '+' 5	40 \$	state 1 factor : "NUM" . (4)	. error
shift	0 e 3 '+' 5 NUM 1	\$. reduce 4	
factor: NUM	0 e 3 '+' 5 factor	\$	state 2 \$accept : s . \$end (0)	state 6 exp : exp '+' factor . (2)
	0 e 3 '+' 5 factor 6	\$	\$end accept	
e: e + factor	0 e	\$	state 3 s : exp . (1)	. reduce 2
	0 e 3	\$	exp : exp . '+' factor (2)	
s: exp	0 s	\$	'+' shift 5	\$end reduce 1
	0 s 2	\$		

Solución E1 (II)

state 0

\$accept : . s \$end (0)

"NUM" shift 1

. error

s goto 2

exp goto 3

factor goto 4

state 1

factor : "NUM" . (4)

. reduce 4

state 2

\$accept : s . \$end (0)

\$end accept

state 3

s : exp . (1)

exp : exp . '+' factor (2)

'+' shift 5

\$end reduce 1

state 4

exp : factor . (3)

. reduce 3

state 5

exp : exp '+' . factor (2)

"NUM" shift 1

. error

factor goto 6

state 6

exp : exp '+' factor . (2)

. reduce 2

	+	NUM	\$		s	e	f
0		s1			2	3	4
1	r4	r4	r4				
2			ac				
3	s5		r1				
4	r3	r3	r3				
5		s1					6
6	r2	r2	r2				

s: exp (1)

exp: exp '+' factor (2)

| factor (3)

factor: NUM (4)

Solución E2 (I)

Ejemplo de Sentencia

(double) 3 + 4

Opción 1

- Cambiar el lenguaje.

```
%left '+'
%%
expr:  expr '+' expr
      |  LITENT
      |  '(' expr ')'
      |  '(' TYPE ')' '<' expr '>'
```

□ Ejemplos de sentencias

(double)<3> + 4

(double)<3 + 4>

Solución E2 (II)

Opción 2

- Transformar la gramática
 - Hallar una gramática equivalente

```
expr → termino
      | expr + termino
termino → factor
         | '(' TYPE ')' termino
factor → LITENT
        | '(' expr ')'
```

Opción 3

- Reglas de selección

```
%left '+'
%left cast
%%
expr:  expr '+' expr
      | LITENT
      | '(' expr ')'
      | '(' TYPE ')' expr %prec cast
      ;
```