El generador de analizadores sintácticos *Yacc*
III

Teoría de autómatas y lenguajes formales
Alma María Pisabarro Marrón (alma@infor.uva.es)
Universidad de Valladolid

1. Acciones en Yacc

Las acciones en Yacc son, como ya hemos visto, código c encerrado entre llaves asociado a cada regla. Cada acción se ejecuta cuando se aplica durante el análisis la regla a la que esta asociada. Las acciones pueden devolver valores y usar los obtenidos en acciones anteriores, en particular los devueltos por Lex como valores léxicos de los componentes reconocidos.

Ejemplo1: El siguiente ejemplo implanta una gramática de paréntesis anidados y muestra el mensaje "Correcto" cuando se consigue aplicar la primera regla, es decir, cuando se consigue el árbol de derivación completo.

```
%{
#include <stdio.h>
yyerror (char *s) {
      fprintf (stderr, "%s\n", s) ;
}
%}

%%
I     : S '\n' {printf ("Correcto\n");}
      ;
S     : '(' S ')' S
      |
      ;
%%
yylex() { /* código de yylex proporcionado directamente */
      return (getchar()) ; /* devuelve el propio código (ASCII) del
carácter */
}

main() {
      yyparse();
}
```

Ejemplo2: El ejemplo siguiente muestra las reglas empleadas en la derivación. Como el análisis realizado por un analizador construido can Yacc es ascendente las reglas aparcen en orden inverso al empleado. Probando con las entradas `()  ()` y `(())` podrá detectarse que la derivación reflejada es más a la derecha.

```
%{
#include <stdio.h>
yyerror (char *s) {
      fprintf (stderr, "%s\n", s) ;
}
%}
```

```
%%
I     : S '\n'              {printf ("Correcto\n");}
      ;
S     : '(' S ')' S         {printf ("S -> (S)S\n");}
      |                     {printf ("S -> epsilon\n");}
      ;
%%
yylex() { /* código de yylex proporcionado directamente */
      return (getchar()) ; /* devuelve el propio código (ASCII) del
carácter */
}

main() {
      yyparse();
}
```

1.1. *Reglas Semánticas*

Se pueden devolver valores en la parte izquierda de una regla utilizando unas variables especiales de Yacc a la vez que se aplica una regla de derivación. Esto permite mover valores de forma ascendente, a la vez que se realiza el análisis, por el árbol de derivación obtenido para el análisis de una expresión. Estas variables almacenan el "valor semántico" de cada uno de los símbolos de cada regla, estan definidas por Yacc y sus nombre nombres identificadores son invariables. Son las siguientes:

- **`$$`** representa la parte izquierda de una regla
- **`$1, $2`**,... representan a los valores asociados a los símbolos de la parte derecha de una regla. Así `$n` almacena el valor asociado al símbolo que ocupa la posición `n` en la parte derecha de la regla.

Las asignaciones que permiten este movimiento de valores semánticos aparecen como parte de la acción asociada a una regla y se denominan reglas semánticas.

Ejemplo3: El ejemplo siguiente muestra la sección de reglas de un programa Yacc para una gramática que reconoce el lenguaje formado por las expresiones aritméticas de suma y resta y además devuelve el valor obtenido de evaluar la expresión aritmética de entrada.

```
%%
S     : exp '\n'            {printf ("Resultado: %d\n", $1);}
      ;
exp   : exp '+' exp         {$$ = $1+$3}
      | exp '-' exp         {$$ = $1-$3}
      | '(' exp ')'         {$$ = $2}
      | NUM                 {$$ = $1}
      ;
%%
```

2. Asociatividad y Precedencia

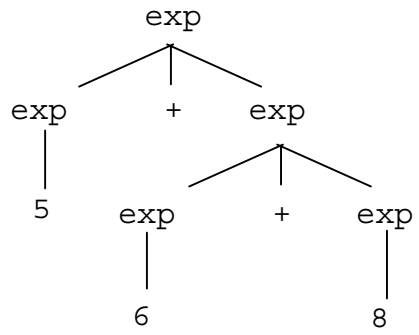
Por defecto Yacc es asociativo a la derecha pero esto puede modificarse en la sección de definiciones utilizando las siguientes directivas:

- **`%left`** asocia a la izquierda
- **`%right`** asocia a la derecha

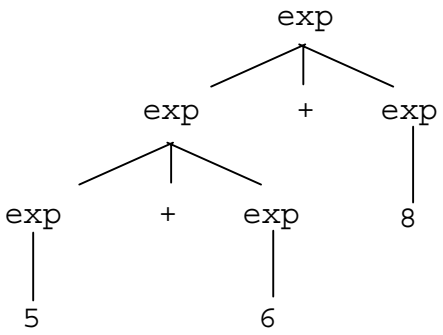
Supongamos el siguiente fragmento de programa Yacc para una gramática que reconoce expresiones de sumas de números:

```
%token NUM
%left '+'
%%
exp    : exp '+' exp          {$$ = $1+$3}
       | NUM                  {SS = $!}
       ;
```

Si la entrada fuese `5+6+8` por defecto se resolvería como `5+(6+8)` (asociación por la derecha), pero al haber definido `%left` al símbolo `'+'` en la sección de definiciones el resultado es `(5+6)+8` (asociación por la izquierda).
Los árboles de análisis obtenidos de una manera o de otra serían bastante diferentes:



Asociación por la derecha



Asociación por la izquierda

`%left, %right` y **`%nonassoc`** Además de indicar asociatividad se utilizan en la sección de definiciones para indicar precedencia de los operadores teniendo mayor precedencia cuanto más alejados estén del principio del fichero. La última de las tres directivas no indica asociatividad solo precedencia.
`%pred` Se utiliza en una regla para dar distinta prioridad al primer símbolo de esa regla, solo para la aplicación de esa regla. Se usa cuando un mismo símbolo puede tener significados distintos con prioridades diferentes. Por ejemplo el símbolo '-' puede ser unario y binario, cuando es binario tiene la misma prioridad que '+', pero cuando es unario tiene mayor prioridad. En este caso se le puede asignar la misma prioridad que a '*', tal y como se muestra en el ejemplo siguiente.

```
%right '='        /*menor precedencia*/
%left '+', '-'
%left '*', '/'    /*mayor precedencia*/
%%
exp    : exp '=' exp
       | exp '+' exp
       | exp '-' exp
       | exp '*' exp
       | exp '/' exp
       | '-' exp  %pred '*'
       | CAR
       ;
```

Así frente a la entrada `a=b=c*d-e-f*g` se analizaría como `a=(b=(((c*d)-e)-(f*g)))`

3. Conflictos y Ambigüedad

Cuando la gramática de entrada no es LALR(1), Yacc informará de que ha encontrado "conflictos". Los conflictos se producen cuando el algoritmo de construcción de las tablas no ha permitido, en un momento dado, que se sepa si la acción adecuada es desplazar o reducir (conflicto `shift/reduce`), o bien cuando haya varias posibilidades de reducción (conflicto `reduce/reduce`).
Se podrá encontrar cuales son los conflictos aparecidos examinando el contenido de `y.output`. A

veces es conveniente conocer cuales son los conflictos que provoca nuestra gramática y tratar de evitarlos, para ello puede ser útil utilizar la opción `-v` de Yacc para examinar el autómata descrito en el fichero `y.output`.
Independientemente de la información mostrada Yacc continua el proceso, generando un programa que resuelve sistemáticamente los conflictos aplicando las siguientes normas:

- Entre desplazar y reducir, elige desplazar.
- Entre dos reducciones distintas, elige la correspondiente a la regla que aparezca en primer lugar

3.1. *Conflicto Reducción-Reducción*

La siguiente especificación, por ejemplo, produce un conflicto reducción-reducción.

```
%token CART PLOW AND HORSE GOAT OX
%%
phrase          : cart_animal AND CART
                | work_animal AND PLOW
                ;
cart_animal     : HORSE
                | GOAT
                ;
work_animal     : HORSE
                | OX
                ;
```

Al llamar a Yacc con este programa fuente aparecerán los siguientes mensajes:

```
yacc:          1 rule never reduced
yacc:          1 reduce/reduce conflict
```

cuyo significado se explica un poco más en el fichero `y.output`.
El problema aparece ante una frase como `HORSE AND PLOW`

PILA	ENTRADA	ACCIÓN
`$`	`HORSE AND PLOW`	desplazar
`$ HORSE`	`AND PLOW $`	reducir ¿por `cart_animal` (regla 3) o `work_animal` (regla 5)? (no se sabe que `HORSE` es un `work_animal` porque aún no se ve `PLOW`)

Obsérvese que en este ejemplo el método de resolución de conflictos de Yacc (en este caso, reducción-reducción, reducir por la primera regla) da lugar a que se considere la frase como incorrecta aún siendo correcta.

La siguiente gramática es equivalente pero LALR(1):

```
phrase          : cart_animal CART
                | work_animal PLOW
                ;
cart_animal     : HORSE AND
                | GOAT AND
                ;
work_animal     : HORSE AND
                | OX AND
                ;
```

3.2. *Conflicto Desplazamiento-Reducción*

Un ejemplo clásico de conflicto desplazamiento/reducción es la provocada por los lenguajes de programación con la estructura condicional simple y compuesta. Por ejemplo supongamos la siguiente gramática:

```
stat   : IF '(' cond ')' stat
       : IF '(' cond ')' stat ELSE stat
       ;
```

Si la entrada fuese `IF (a<3) a=0 ELSE a++` hay conflicto desplazamiento-reducción, se puede desplazar por la regla2 o reducir por la regla1. Se podría reducir por la primera regla una vez leído `IF (a<3) a=0` , lo que sería incorrecto. Yacc continúa para comprobar si puede aplicar la segunda regla (frente a un conflicto desplazar-reducir, Yacc siempre desplaza).

La entrada `IF c1 IF c2 s1 ELSE s2` puede interpretarse como `IF c1 {IF c2 s1} ELSE s2` o como `IF c1 {IF c2 s1 ELSE s2}` que es la más usada en lenguajes de programación, se toma el `ELSE` como asociado al `IF` más cercano. Comprobar como lo resuelve Yacc.

Otro ejemplo de conflicto desplazamiento-reducción lo tenemos en la gramática siguiente:

```
%token DIG
%%
E      : E '+' E
       | DIG
       ;
```

En este caso la gramática es ambigua y eso produce los conflictos como podemos ver en las tablas de desplazamiento-reducción siguientes:

ACCIÓN		DIG	+	$end	**Ir_A**	E
estado	significado					
0	`$accept : . E $end (0)`	shift 1				2
1	`E : DIG . (2)`	reduce 2	reduce 2	reduce 2		
2	`$accept : E . $end (0)` `E : E . '+' E (1)`		shift 3	accept		
3	`E : E '+' . E (1)`	shift 1				4
4	`E : E . '+' E (1)` `E : E '+' E . (1)`		shift 3 (reduce 1)	reduce1		

Comprobar cómo resuelve Yacc los conflictos en este caso.