Gramáticas con atributos.

Introducción y definiciones

A diferencia del análisis sintáctico, en donde en la actualidad existe un marco teórico ampliamente aceptado, el tema del análisis semántico suele ser tratado de diversas formas. La teoría que seguiremos para la formulación del análisis semántico y la posterior generación de código se basa en las gramáticas atribuidas.

Antes de dar una definición formal intentaremos dar una idea intuitiva de las mismas.

Una gramática atribuida, no es más que una gramática de contexto libre a cuyos símbolos `x` (terminales y no terminales) asociamos un conjunto de atributos. Cada atributo es como una variable que representa una determinada propiedad del símbolo `x`, y puede tomar un valor cualquiera de un determinado conjunto de valores posibles. Denotaremos a este atributo con un nombre precedido del símbolo al que esta asociado: `x.a`, `x.valor`, `x.tipo`, etc.... Al conjunto de todos los atributos asociados al símbolo `x` se le llama *conjunto de atributos* de `x`, que denotaremos por `A(X)`.

Ejemplo: Sea la gramática que representa las asignaciones

```
(1) ASIGNACION --> VARIABLE igual EXPRESION(2) EXPRESION  --> EXPRESION OPERADOR nume
```

Esta gramática genera el lenguaje que contiene las siguientes cadenas de tokens:

identificador igual numero
identificador igual numero mas numero
identificador igual numero mas numero mas numero etc ...

Estas cadenas de tokens son formadas por el analizador lexicográfico a partir de un fichero de entrada que contendría los lexemas correspondientes como ya vimos en el capítulo dedicado a analizadores lexicográficos. Por ejemplo:

Cuatro := 2.0 + 2
Alfa := 3 + 4.5 + 1.5

Vamos a considerar los atributos siguientes, cada uno de los cuales representa una propiedad determinada del símbolo al que esta asociado.

numero.tipo	Tipo del número (entero o real)
numero.valor	Valor que el número representa
identificador.tipo	Tipo del identificador. Es decir, tipo con el que dicho identificador ha sido definido
variable.tipo	Tipo de la variable que interviene en la sentencia de asignación
variable.valor	Valor que toma dicha variable
expresion.tipo	Tipo de una expresión

expresion.valor	Valor que obtenemos al evaluarla
operador.tipo	Clase de operación (suma entera o real)

Los atributos asociados a `x` tomarán una serie de valores determinados en cada nodo del árbol sintáctico de una sentencia del lenguaje generado por la gramática en el que esté situado el símbolo `x`. Estos valores serán establecidos de acuerdo con un conjunto de reglas semánticas asociadas a alguna de las reglas sintácticas de la gramática de las que intervengan en el nodo. Al conjunto de reglas semánticas asociado a una regla sintáctica lo llamaremos `R(p)`. Las reglas semánticas vienen dadas en función de los atributos de los demás símbolos que componen la regla.

Ejemplo: Continuando con el ejemplo anterior, vamos a formar el arbol sintáctico correspondiente al análisis de la cadena de caracteres :

Cuatro := 2.0 + 2

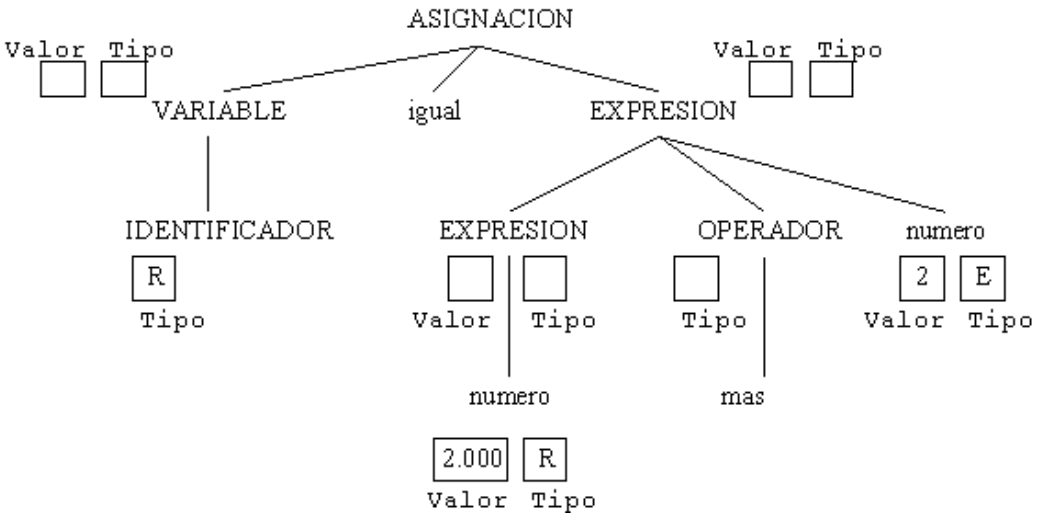
Tras realizar el análisis lexicográfico obtenemos la cadena de tokens:

identificador igual numero mas numero

Además, el analizador lexicográfico ha realizado las correspondientes acciones semánticas y nos da los valores de algunos de los atributos de los tokens (símbolos terminales de nuestro análisis). Estos atributos, como ya veremos, reciben el nombre de *atributos intrínsecos*. Sean estos :

identificador.tipo = Real
numero.tipo = Real
numero.valor = 2.000
numero.tipo = Entero
numero.valor = 2

Cada nodo del árbol estará ocupado por un símbolo de la gramática. Aquellos símbolos que tengan atributos asociados los representaremos con ellos:



Los conjuntos de reglas semánticas asociadas a cada regla sintáctica serían:

(1) ASIGNACION --> VARIABLE igual EXPRESION

VARIABLE.valor := EXPRESION.valor

(2) EXPRESION --> EXPRESION OPERADOR numero

OPERADOR.tipo:=Mayor_tipo(EXPRESION(2).tipo,numero.tipo)

```
EXPRESION(1).valor:=Suma(mas.tipo,EXPRESION(2).tipo,              numero.tipo,EXPRESIC
```

EXPRESION(1).tipo:=OPERADOR.tipo

(3) EXPRESION --> numero

EXPRESION.valor:=numero.valor

EXPRESION.tipo:=numero.tipo

(4) VARIABLE --> identificador

VARIABLE.tipo:=identificador.tipo

(5) OPERADOR --> mas

(No tiene ninguna regla asociada)

En donde la función `Mayor_tipo` se define como:

```
function Mayor_tipo (tipo1,tipo2) ---> tipobegin      if (tipo1=Real or tipo2=Real) t
```

Y la función `Suma` como:

```
function Suma (tipo_suma,tipo1,tipo2,valor1,valor2)              --->valorbegin
```

Vamos a suponer que las funciones `Suma_real` y `Suma_Entera`, y los procedimientos `Ent_Real` y `Real_Ent` (que convierten un entero en un real, y viceversa) pertenecen ya al nivel inferior que es capaz de ejecutar la máquina.

Por lo general, pueden intervenir dos reglas en un nodo de un árbol sintáctico de una cadena del lenguaje, ocupado por el símbolo `X`:

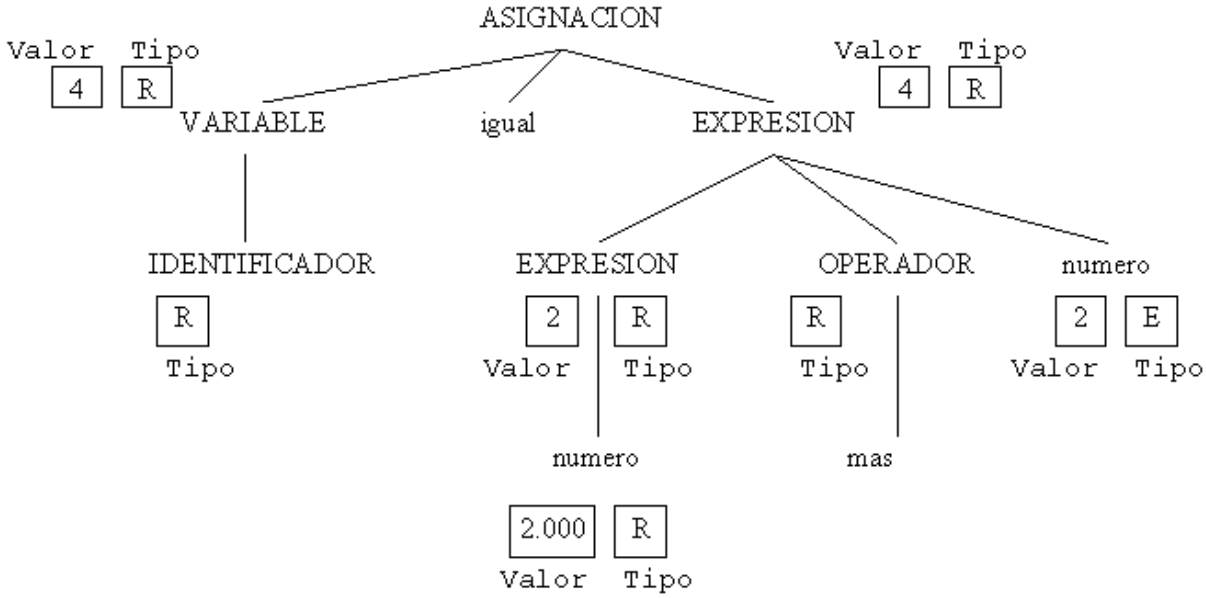
```
S => ..... => αYβ => ατX''β => αττ''β => ..... t                (p)      (q)

p:    Y ---> τX''q:    X ---> τ
```

Algunos de los valores de los atributos del símbolo `X` en dicho nodo se calculan mediante reglas semánticas asociadas a la regla anterior `(p)`, mientras que otros se calculan mediante reglas asociadas a la regla posterior `(q)`. En el primero de los casos diremos que el atributo correspondiente a `X` es *heredado*, mientras que en el segundo caso diremos que el atributo es *sintetizado*. Al conjunto de atributos heredados lo llamaremos `AH(X)`, y al conjunto de atributos derivados `AS(X)`. Los conjuntos `AH(X)` y `AS(X)` **son disjuntos**.

Por último, y para garantizar la corrección semántica de una sentencia se establecen unas condiciones que han de cumplir los atributos de cada uno de los símbolos de una misma regla sintáctica. LLamamos a este *conjunto de condiciones* `B(p)`.

Ejemplo: Continuando con el ejemplo anterior, aplicando las reglas correspondientes en cada caso podemos calcular los atributos de todos los símbolos en todos los nodos del árbol sin más que aplicar la regla semántica que necesitemos. Nótese que sólo una regla semántica es aplicable para el cálculo de un determinado atributo de un símbolo en un nodo. El árbol sintáctico quedaría:



Dependiendo de la producción a la que esté asociada la regla que empleamos para calcular los atributos, podemos clasificar a estos en *atributos sintetizados* y *atributos heredados*. Se va a considerar que los símbolos terminales sólo tienen atributos sintetizados, que son proporcionados por el analizador lexicográfico. Los siguientes atributos son sintetizados:

EXPRESION.tipo

EXPRESION.valor

VARIABLE.tipo

numero.tipo

numero.valor

identificador.tipo

Y los siguientes serían atributos heredados:

VARIABLE.valor

OPERADOR.tipo

Por último, podemos establecer algunas condiciones para que las sentencias analizadas sean correctas. En este caso, sólo impondremos una condición asociada a la primera regla:

$$B(1) := (VARIABLE.tipo = EXPRESION.tipo)$$

A continuación vamos a definir formalmente el concepto de gramática atribuida:

Definición: Una *gramática atribuida* es una cuádrupla `GA={G,A,R,B}`, en donde:

- `G={N,T,P,S}` es una gramática de contexto libre.
- `A=U A(X)` es el conjunto de atributos asociados a cada símbolo `X` de la gramática (terminal o no terminal), tales que cumplen que: "para todo símbolo `X` e `Y` (terminal o no terminal) `A(X)^A(Y)=0`". Es decir, los atributos de cada símbolo son disjuntos.

- Dado un símbolo `X`, vamos a denotar los atributos asociados a él como `X.a`, `X.b`, etc..., o con las letras minúsculas `v`,`w` para casos generales.
- `R=U R(p)` es un conjunto finito de reglas, o acciones semánticas, asociadas a cada una de las reglas sintácticas de la gramática `G`. Sea la regla p de la forma:

  $$X_0 \rightarrow X_1X_2...X_n$$

  Entonces, las reglas semánticas asociadas a esta regla sintáctica son de la forma `v:=` $f(w_1,w_2,...,w_k)$, en donde:
  - `f` es una función.
  - `v`,$w_1,...,w_n$ pertenecen a `U A(`X_i`)`. Es decir, son atributos de los símbolos que componen la regla `p`.
- `B=U B(p)` es un conjunto finito de condiciones asociadas a cada una de las reglas sintácticas de la gramática `G`. Sea la regla `p` de la forma X_0 `->` $X_1X_2...X_n$. Entonces, las condiciones son de la forma $g(w_1,w_2,...w_k)$, donde:
  - `g` es una función booleana
  - $w_1,w_2,...,w_k$ pertenecen a `U A(`X_i`)`

Cuando las funciones de las reglas semánticas pueden tener efectos laterales, tales como imprimir un valor, introducir una variable en una tabla de símbolos, etc..., la gramática atribuida pasa a ser una *definición atribuida*, también denominada *definición dirigida por sintaxis*.

Atributos sintetizados y heredados: Si `p` es una regla sintáctica de la forma:

$$p: X_0 \rightarrow X_1X_2...X_n$$

entonces se define el conjunto de *atributos calculados* `AC(p)` mediante la regla `p`, como:

$$AC(p) = \{X_i.a \ / \ X_i.a = f\ (...) \text{ pertenece a } R(p)\}$$

Dada una gramática atribuida, se dice que un atributo asociado a un símbolo `X` es *sintetizado* si existe una regla sintáctica de la forma `X -> τ y` una regla semántica que lo calcula a partir de los atributos de los símbolos del consecuente de dicha regla. Al conjunto de los atributos sintetizados asociados a `X` lo denotaremos mediante `AS(X)`.

```
AS(X)={X.a / Existe X -> τ perteneciente a P y            X.a pertenece a AC(p)}
```

Dada una gramática atribuida, se dice que un atributo asociado a un símbolo `X` es *heredado* si existe una regla sintáctica de la forma `Y -> αXβ`, y una regla semántica que lo calcula a partir de los atributos de los demás símbolos que forman la regla. Al conjunto de los atributos heredados asociados a `X` lo denotaremos mediante `AH(X)`.

```
AH(X)={X.a / Existe Y -> αXβ perteneciente a P y            X.a pertenece a AC(p)}
```

Gramatica atribuida completa: Se dice que una gramática atribuida es *completa* si se cumplen las cuatro condiciones siguientes para todo símbolo `X`.

1. Para toda regla `p: X -> τ`, `AS(X)` está incluido en `AC(p)`.

2. Para toda regla `q: Y -> αXβ`, `AH(X)` está incluido en `AC(q)`.

3. `AS(X) U AH(X) = A(X)`

1. `AS(X)^AH(X) = 0`

De estas condiciones se deduce que el axioma de la gramática (`S`) no puede tener más que atributos sintetizados (es decir, `AH(S)=0`). Los atributos de los símbolos terminales se consideran como sintetizados (tambien se les denomina *atributos intrínsecos*, ya que tienen un valor en sí mismos y no es necesario calcularlo)

Gramáticas atribuidas bien definidas: Una gramática atribuida se dice que está *bien definida* si para toda sentencia de `L(G)` todos los atributos pueden ser computados. A este proceso se le denomina *evaluación de la gramática*. Una sentencia del lenguaje se dice que está *correctamente atribuida* si se verifican todas las condiciones `B(p)` asociadas a las reglas que la generan.

Teorema: Toda gramática atribuida bien definida es completa.

Gramáticas con atributos.

Grafos de dependencia

Grafo de dependencias directas: Se define el grafo de dependencias directas de una regla $X_1X_2...X_n$, como un conjunto de pares ordenados $(v,w)=(X_i.a,X_j.b)$

$$GDD(p): \{(v,w) / v := f(w_1,w_2,...,w_k) \text{ pertenece a } R(p) \quad \text{y } w \text{ pertenece a } \{wi\}\}$$

Se dice que la gramática atribuida es *localmente acíclica* si para toda regla p el grafo de dependencias directas es acíclico.

Grafo de dependencias: Dada una sentencia s del lenguaje, cuyo árbol sintáctico está formado por una secuencia de reglas $p_1,p_2,...,p_m$, y cuyos nodos son $K_0,K_1,...,K_n$, en cada uno de los cuales existe un símbolo de la gramática, (al que llamaremos del mismo modo)

El grafo de dependencias ($GD(s)$) de la sentencia s se define como el conjunto de pares ordenados tales que:

$$GD(s)=\{(K_i.a,K_j.b) / \text{Existe } p_k \text{ perteneciente a } \{p_1,...,p_m\} \quad \text{y } (K_i.a,K_j.b) \text{ pertenece a } GDD(p_k)\}$$

Es decir, el grafo de dependencias es la fusión de todos los grafos de dependencias directas de las reglas $p_1,p_2,...,p_n$, que generan su árbol sintáctico.

Teorema: Una gramática atribuida está *bien definida* si y sólo si es completa, y para toda sentencia s del lenguaje $L(G)$ el grafo de dependencias $GD(s)$ es acíclico.

En una gramática atribuida bien definida existe un algoritmo capaz de calcular los valores de todos los atributos en todos los nodos del árbol sintáctico sea cual sea la cadena de entrada.

Desgraciadamente no existe ningún algoritmo que en un número finito de pasos calcule si una gramática atribuida está bien definida al ser ésta una propiedad indecidible en el caso general. Usaremos, por tanto, subclases de gramáticas atribuidas para las que sí podamos determinar esta propiedad.

Gramáticas con atributos.

Gramáticas S-atribuídas

Se dice que una gramática es *S-atribuida* si sólo contiene atributos sintetizados.

Ejemplo: Un ejemplo clásico de gramática S-atribuida es la gramática de las expresiones aritméticas:

S -> E
E -> E+T
E -> T
T -> T*F
T -> F
F -> (E)
F -> a

con los atributos `S.val`, `E.val`,`T.val`,`F.val` y `a.val`, y las reglas semánticas:

(1) S -> E	S.val := E.val
(2) E -> E+T	E.val := E1.val+T.val
(3) E -> T	E.val := T.val
(4) T -> T*F	T.val := T1.val*F.val
(5) T -> F	T.val := F.val
(6) F -> (E)	F.val := E.val
(7) F -> a	F.val := a.val

Gramáticas con atributos.

Esquemas de traducción atribuiídos.

Esquema de traducción atribuido. Un *esquema de traducción atribuido* es una gramática atribuida en la cuál las reglas semánticas aparecen intercaladas entre los símbolos del consecuente de las producciones o reglas sintácticas, indicando así el momento en que deben ser ejecutadas.

Las producciones de un esquema de traducción semántico son de la forma:

$$A \rightarrow \{ R_0 \} X_1 \{ R_1 \} X_2 \ldots X_n \{ R_n \}$$

En donde R_i representa una o varias acciones semánticas del conjunto de acciones asociado a la regla.

Esquemas de traducción atribuidos bien definidos. Para que un esquema de traducción atribuido esté bien definido debe cumplir las siguientes condiciones (condiciones suficientes):

1. Un atributo heredado asociado a un símbolo en el consecuente de una regla debe ser evaluado antes de que se emplee la producción correspondiente al mismo.

2. Una regla semántica asociada a una producción sintáctica no debe hacer referencia a los atributos sintetizados asociados al antecedente de dicha producción.

3. Un atributo sintetizado asociado a un símbolo no terminal situado en el antecedente de una regla debe ser evaluado después de que hayan sido evaluados todos los atributos de los que él dependa.

Con estas tres condiciones garantizamos que es posible realizar una correcta evaluación del esquema de traducción, y por tanto que la gramática atribuida subyacente esta bien definida. Por otra parte, puede demostrarse que a partir de una gramática L-atribuida es posible obtener siempre un esquema de traducción que satisfaga estas tres condiciones. Por todo ello podemos afirmar que **toda** gramática L-atribuida está bien definida.

Es de resaltar también el hecho de que un esquema de traducción bien definido asociado a una gramática S-atribuida, como consecuencia de las tres citadas anteriormente, tiene solamente reglas de la forma:

$$A \rightarrow X_1 X_2 \ldots X_n \{ R_n \} \; R_n ::= \{ A.s_k := f(X_i.s_j) \}$$

Gramáticas con atributos

YACC

YACC significa "otro compilador de compiladores más" (del inglés *Yet Another Compilers-Compiler*), lo que refleja la popularidad de los generadores de analizadores sintácticos al principio de los años setenta, cuando S. C. Johnson creó la primera versión de YACC. Este generador se encuentra disponible como una orden del sistema UNIX, y se ha utilizado para facilitar la implantación de cientos de compiladores.

Para construir un traductor utilizando YACC primero se prepara un archivo, por ejemplo `traduce.y`, que contiene una especificación en YACC del traductor. La orden del sistema UNIX

```
yacc traduce.y
```

transforma al archivo `traduce.y` en un programa escrito en C llamado `y.tab.c`, que implementa un analizador sintáctico escrito en C, junto con otras rutinas en C que el usuario pudo haber preparado en el fichero `traduce.y`. Posteriormente, se compila el fichero `y.tab.c` y se obtiene el programa objeto deseado que realiza la traducción especificada por el programa original en YACC. Si se necesitan otros procedimientos, se pueden compilar o cargar con `y.tab.c`, igual que en cualquier programa en C.

Un programa fuente en YACC tiene tres secciones:

declaraciones
%%
reglas de traducción
%%**rutinas en C de apoyo**

- **La parte de declaraciones**. Hay dos secciones opcionales en la parte de declaraciones de un programa en YACC:
  - En la primera sección, se ponen declaraciones ordinarias en C, delimitadas por `%{` y `%}`. Aquí se sitúan las declaraciones de todas las variables temporales usadas por las reglas de traducción o los procedimientos de la segunda y tercera secciones. Por ejemplo:

    ```
    %{     #include <string.h>     %}
    ```

    También en la parte de declaraciones hay declaraciones de los componentes léxicos de la gramática. Por ejemplo:

    %token DIGITO

    declara que `DIGITO` es un componente léxico o token. Los componentes léxicos que se declaran en esta sección se pueden utilizar después en la segunda y tercera partes de la especificación en YACC.

- **La parte de las reglas de traducción**. En la parte de la especificación en YACC después del primer par %% se escriben las reglas de traducción. Cada regla consta de una producción de la gramática y la acción semántica asociada. Un conjunto de producciones como:

  < lado izquierdo > -> < alt1 > | < alt2 > ... | < altn >

  En YACC se escribiría:

  ```
  < lado izquierdo >  : < alt1 >      { acción semántica 1 }                    | < alt2 >      { acción semántica 2 }
  ```

  En una producción en YACC, un carácter simple entrecomillado 'c' se considera como el símbolo terminal `c`, y las cadenas sin comillas de letras y dígitos no declarados como componentes léxicos se consideran símbolos no terminales. Los lados derechos alternativos de las reglas se pueden separar con una barra vertical, y un símbolo de punto y coma sigue a cada lado izquierdo con sus alternativas y sus acciones semánticas. El primer lado izquierdo se considera, por defecto, como el símbolo inicial.

  Una acción semántica en YACC es una secuencia de sentencias en C. En una acción semántica, el símbolo `$$` se refiere al valor del atributo asociado con el no terminal del lado izquierdo, mientras que `$i` se refiere al valor asociado con el i-ésimo símbolo gramatical (terminal o no terminal) del lado derecho. La acción semántica se realiza siempre que se reduzca por la producción asociada, por lo que normalmente la acción semántica calcula un valor para `$$` en función de los `$i`. Si la regla no especifica ninguna acción semántica, la acción semántica por omisión es `{$$ = $1;}`.

- **La parte de las rutinas de apoyo en C**. La tercera parte de una especificación en YACC consta de rutinas de apoyo escritas en C. Para que el analizador sintáctico funcione, se debe proporcionar un análisis léxico de nombre `yylex()`. En caso necesario se pueden agregar otros procedimientos, como rutinas de recuperación de errores.

  El analizador léxico `yylex()` produce pares formados por un componente léxico y su valor de atributo asociado. Si se devuelve un componente léxico como `DIGITO`, el componente léxico se debe declarar en la primera sección de la especificación en YACC. El valor del atributo asociado a un componente léxico se comunica al analizador sintáctico mediante una variable especial que se denomina `yylval`.

Uso de YACC con gramáticas ambiguas. Si la gramática de la especificación en YACC es ambigua se producen conflictos en las acciones del analizador sintáctico. YACC informará del número de conflictos en las acciones del análisis sintáctico que se produzcan. Se puede obtener una descripción de los conjuntos de elementos y de los conflictos en las acciones de análisis sintáctico invocando a YACC con la opción **-v**. Esta opción generará un archivo adicional llamado `y.output` que contiene los núcleos de los conjuntos de elementos encontrados por el analizador sintáctico, una descripción de los conflictos en las acciones del análisis, y una representación legible de la tabla de análisis sintáctico **LR** que muestra cómo se resolvieron los conflictos de las acciones del análisis sintáctico.

A menos que se ordene lo contrario, YACC resolverá todos los conflictos en las acciones del análisis sintáctico utilizando las dos reglas siguientes:

1. Un conflicto de **reducción/reducción** se resuelve eligiendo la regla en conflicto que se haya escrito primero en la especificación.
2. Un conflicto de **desplazamiento/reducción** se resuelve en favor del desplazamiento.

Como estas reglas que se siguen por omisión, no siempre reflejan lo que quiere el escritor del compilador, YACC proporciona un mecanismo general para resolver los conflictos de *desplazamiento/reducción*. En la parte de declaraciones, se pueden asignar precedencias y asociatividades a los terminales. La declaración

%left '+' '-'

hace que `'+'` y `'-'` tengan la misma precedencia y que sean asociativos por la izquierda. Se puede declarar que un operador es asociativo por la derecha diciendo

%right '^'

y se puede obligar a un operador a ser un operador binario no asociativo (por ejemplo, dos casos del operador no se pueden combinar en absoluto) diciendo:

%nonassoc '<'

A los componentes léxicos se les dan precedencias en el orden en que aparecen en la parte de declaraciones, siendo los primeros los de menor precedencia. Los componentes léxicos de una misma declaración tienen la misma precedencia. Así, la declaración

%right MENOSU

daría al componente léxico `MENOSU` un nivel de precedencia mayor que a los terminales declarados anteriormente.

YACC resuelve los conflictos de *desplazamiento/reducción* asociando una precedencia y asociatividad a cada producción implicada en un conflicto, así como a cada terminal implicado en un conflicto. Si debe elegir entre *desplazar* el símbolo de entrada `a` y *reducir* por la producción `A->a`, YACC reduce si la precedencia de la producción es mayor que la de `a` o si las precedencias son las mismas y la asociatividad de la regla es `%left`. De lo contrario se elige la acción de *desplazar*.

Generalmente, la precedencia de una producción se considera igual a la del símbolo terminal situado más a la derecha. En la mayoría de los casos, esta es la decisión sensata. En las situaciones en que el símbolo terminal situado más a la derecha no proporcione la precedencia adecuada a una producción, se puede forzar una precedencia añadiendo al final a la regla la etiqueta

%prec < terminal >

Entonces, la precedencia y asociatividad de la producción serán las mismas que las de `terminal`, que se define seguramente en la sección de declaraciones. YACC **no** informa de los conflictos de *desplazamiento/reducción* que se resuelven utilizando este mecanismo de precedencia y asociatividad.

Este "`terminal`" puede ser simplemente un marcador. Es decir, el terminal no es devuelto por el analizador léxico, sino que se declara tan sólo para definir una precedencia para una producción. Por ejemplo,

```
expr    : '-' expr %prec MENOSU
```

hace que la regla anterior tenga la precedencia correspondiente al token `MENOSU`, en lugar de la del token `'-'`.