Lección 3: Análisis Semántico

1. Expresiones y Asignación
2. Estructuras de Control
3. Procesamiento de Declaraciones
4. Procedimientos y Funciones
5. Perspectiva

Lecturas: Cooper, capítulo 4
Scott, capítulo 4
Aho, capítulo 8
Fischer, capítulos 10, 11 , 12 , 13, 14
Holub, capítulo 6
Bennett, capítulo 4, 10

1. Expresiones y Asignación

¿ `5+2*4` ?

¿ `5-2-4` ?

¿ `a = b = c` (en C)?

¿ `a ** b ** c` (en ADA)?

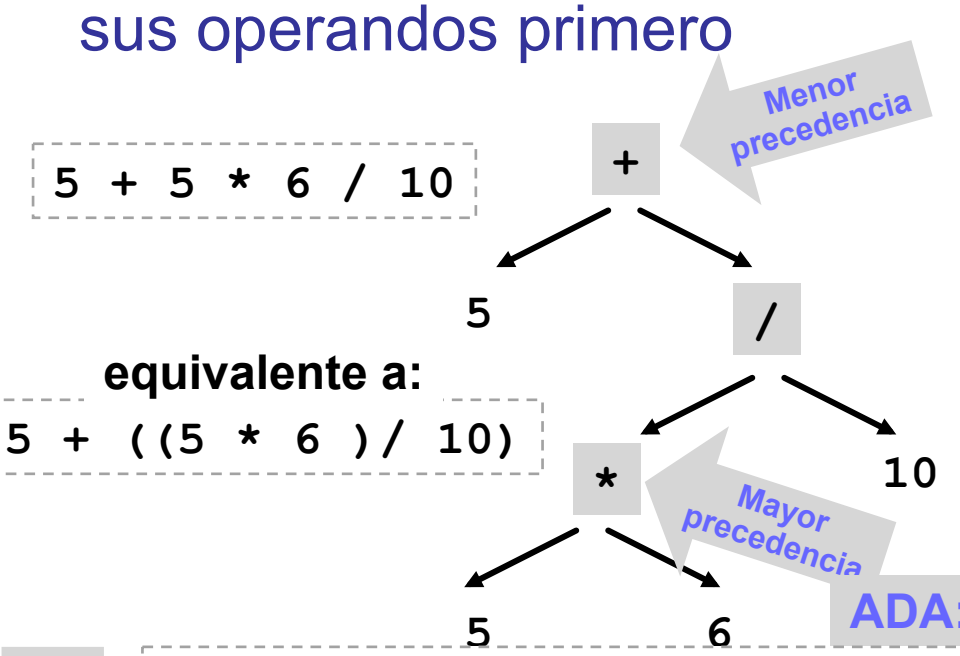
¿ `a < b < c` (en C)?

¿ `a > 0 and b > 0` (en PASCAL)?

Operadores

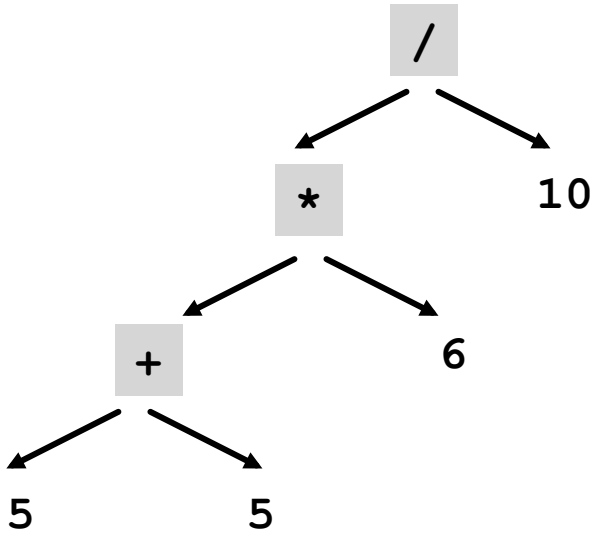
- **Precedencia**: prioridad de los operadores; determina qué ***operadores*** obtienen sus operandos primero
- Se altera con paréntesis

```
((5 + 5) * 6) / 10
```



equivalente a:

```
5 + ((5 * 6 )/ 10)
```



ADA:

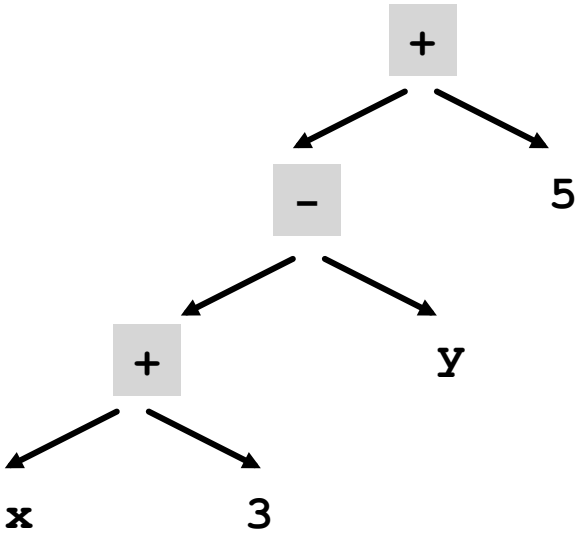
Mayor precedencia

```
logical_operator              ::= and | or | xor
relational_operator           ::= = | /= | < | <= | > |
>=
binary_adding_operator        ::= + | - | &
unary_adding_operator         ::= + | -
multiplying_operator          ::= * | / | mod | rem
highest_precedence_operator   ::= ** | abs | not
```

Operadores

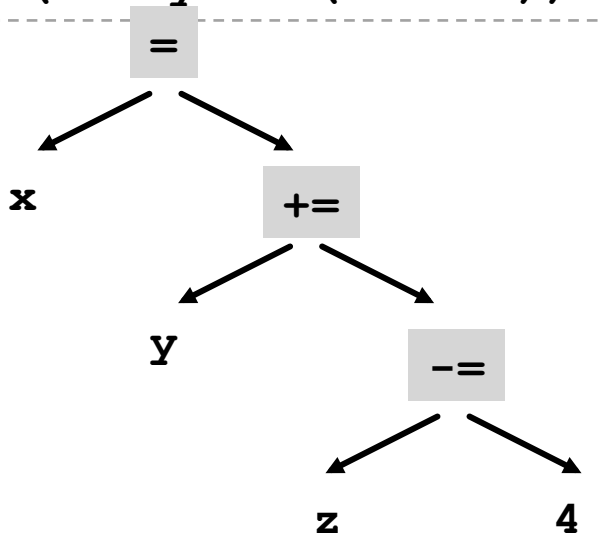
- **Asociatividad:** determina el orden de evaluación de ***operadores*** de igual precedencia
- De izquierda a derecha:

```
x + 3 - y + 5
(x + 3) - y + 5
((x + 3) - y) + 5
(((x + 3) - y) + 5)
```



- De derecha a izquierda

```
x = y += z -= 4
x = y += (z -= 4)
x = (y += (z -= 4))
(x = y += (z -= 4))
```



- Algunos **NO** son asociativos: **ADA:**

```
A**B**C
```

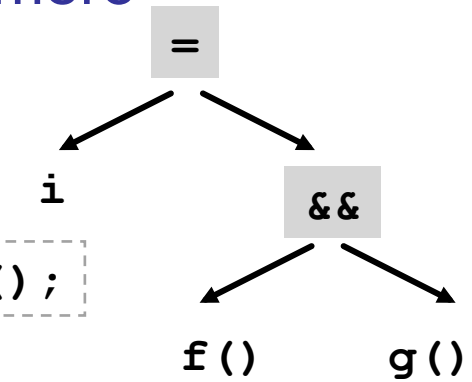
```
(A**B)**C
```

```
A**(B**C)
```

Operadores

- **Orden de evaluación:** determina qué ***operandos*** se evalúan primero

```
i = f() && g();
```



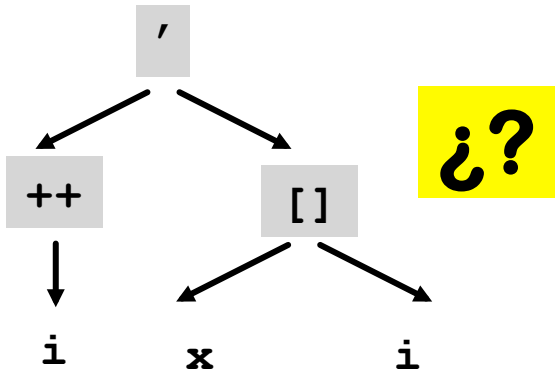
C:

Operator Name	Associativity	Operators
Primary	left to right	() [ ] . ->
Unary	right to left	++ -- + -
Multiplicative	left to right	* / %
Additive	left to right	+ -
Bitwise Shift	left to right	<< >>
Relational	left to right	< > <= >=
Equality	left to right	== !=
Bitwise AND	left to right	&
Bitwise Exclusive OR	left to right	^
Bitwise Inclusive OR	left to right	|
Logical AND	left to right	&&
Logical OR	left to right	||
Conditional	right to left	? :
Assignment	right to left	= += -= *= /= <<= >>=
Comma	left to right	,

Mayor precedencia

- En algunos casos **NO** está definido

```
func2(++i, x[i]);
```



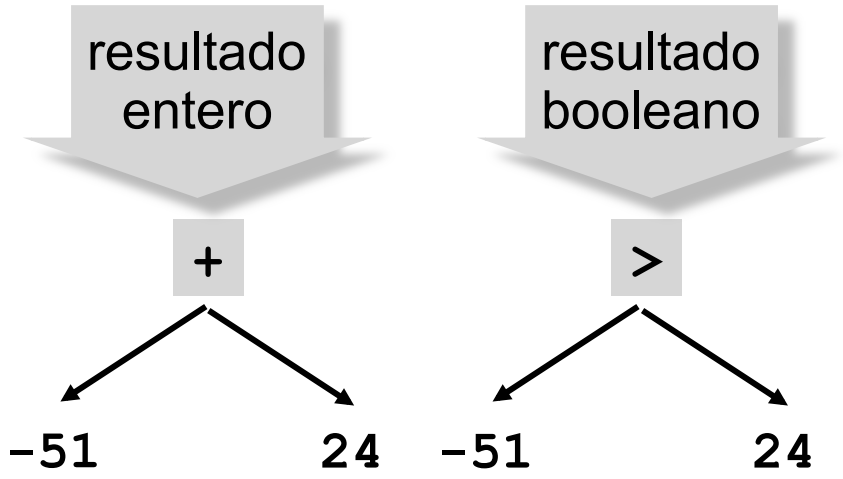
¿?

```
x > 9 && y != 3
x == 5 || y != 3
! x > 14
! (x > 9 && y != 23)
x <= 1 && y == 6 || z < 4
c >= 'a' && c <= 'z'
c >= 'A' || c <= 'Z'
c != d && c != '\n'
5 && y != 8 || 0
x >= y >= z
```

```
int x = 11;
int y = 6;
int z = 1;
char c = 'k';
char d = 'y';
```

Comprobación de expresiones con YACC

```
expr | expr '+' expr
...
     | expr tAND expr
...
     | expr tMAY expr
...
     | '(' expr ')'
...
     | tIDENTIFICADOR
...
     | tCONSTENTERA
  ;
```



- Se utilizan **atributos sintetizados**
- El atributo será el tipo de la expresión
- La producción se valida a partir de los atributos del RHS (`$n`)
- El atributo del LHS (`$$`) se genera a partir de los atributos del RHS

Comprobación de expresiones con YACC

- Constantes e identificadores:

```
%{
typedef enum {
   DESCONOCIDO, ENTERO, BOOLEANO, CHAR, CADENA
   } TIPO_VARIABLE;
%}
%union {
   TIPO_VARIABLE tipo;
   SIMBOLO *simbolo;
}
%type <tipo> expresion
%type <simbolo> tIDENTIFICADOR
%%
expresion : ...
          | tCONSTENTERA
            {
              $$ = ENTERO;
            }
          | tIDENTIFICADOR
            {
              $$ = DESCONOCIDO;
              if ($1 == NULL)
                 error_semantico("identificador desconocido.");
              ...
```

registro de atributos

declaración de atributo por símbolo

atributo LHS sintetizado

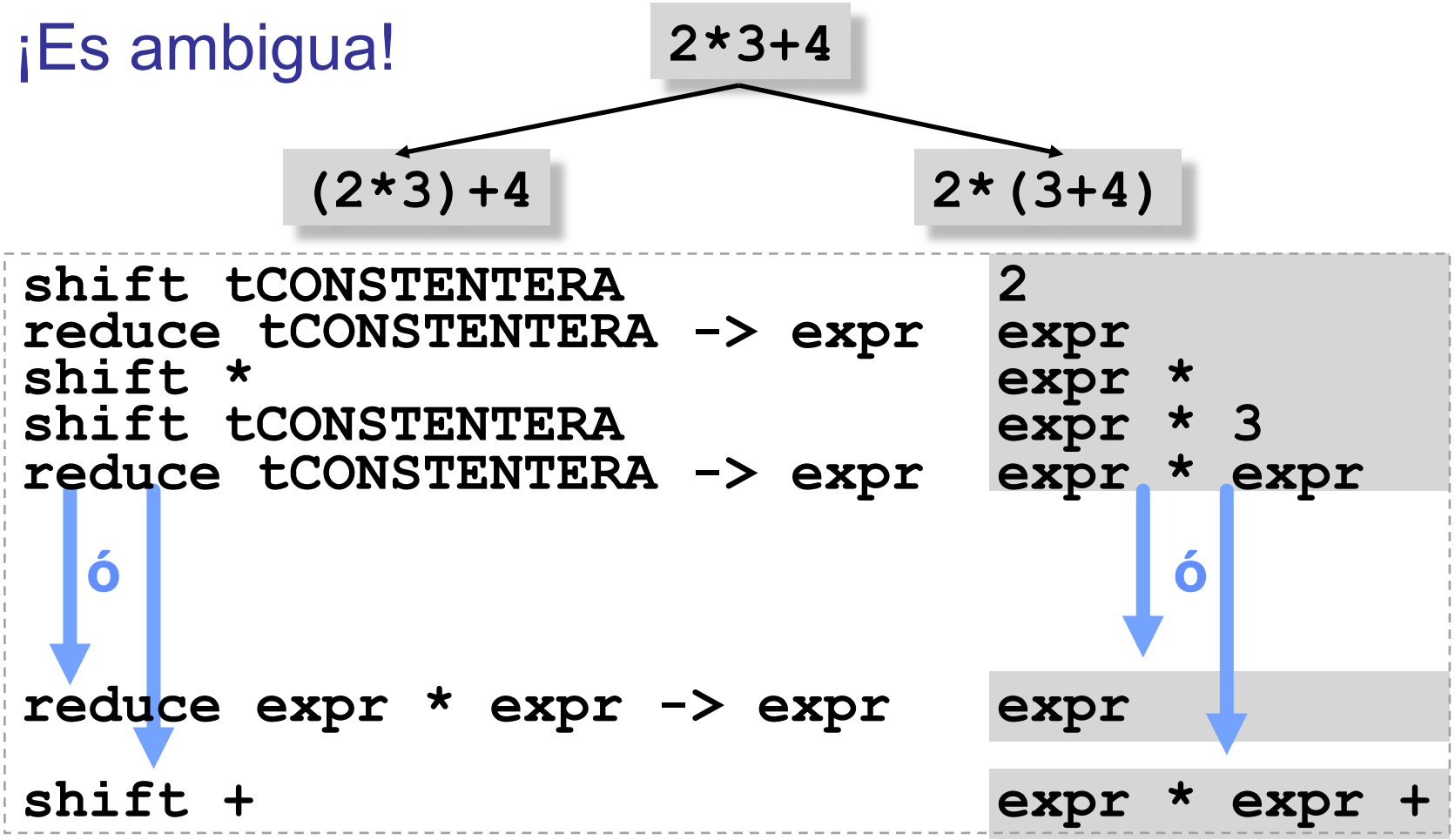
atributo RHS

Comprobación de expresiones con YACC

```
expresion : expresion '+' expresion
          {
            $$ = ENTERO;
            if (($1 != ENTERO) && ($1 != DESCONOCIDO))
               error_semantico("Op 1 debe ser entero.");
          else if (($3 != ENTERO) && ($3 != DESCONOCIDO))
               error_semantico("Op 2 debe ser entero.");
          }
          | expresion tAND expresion
          ...
          | expresion '=' expresion
;
```

YACC: comprobación de expresiones

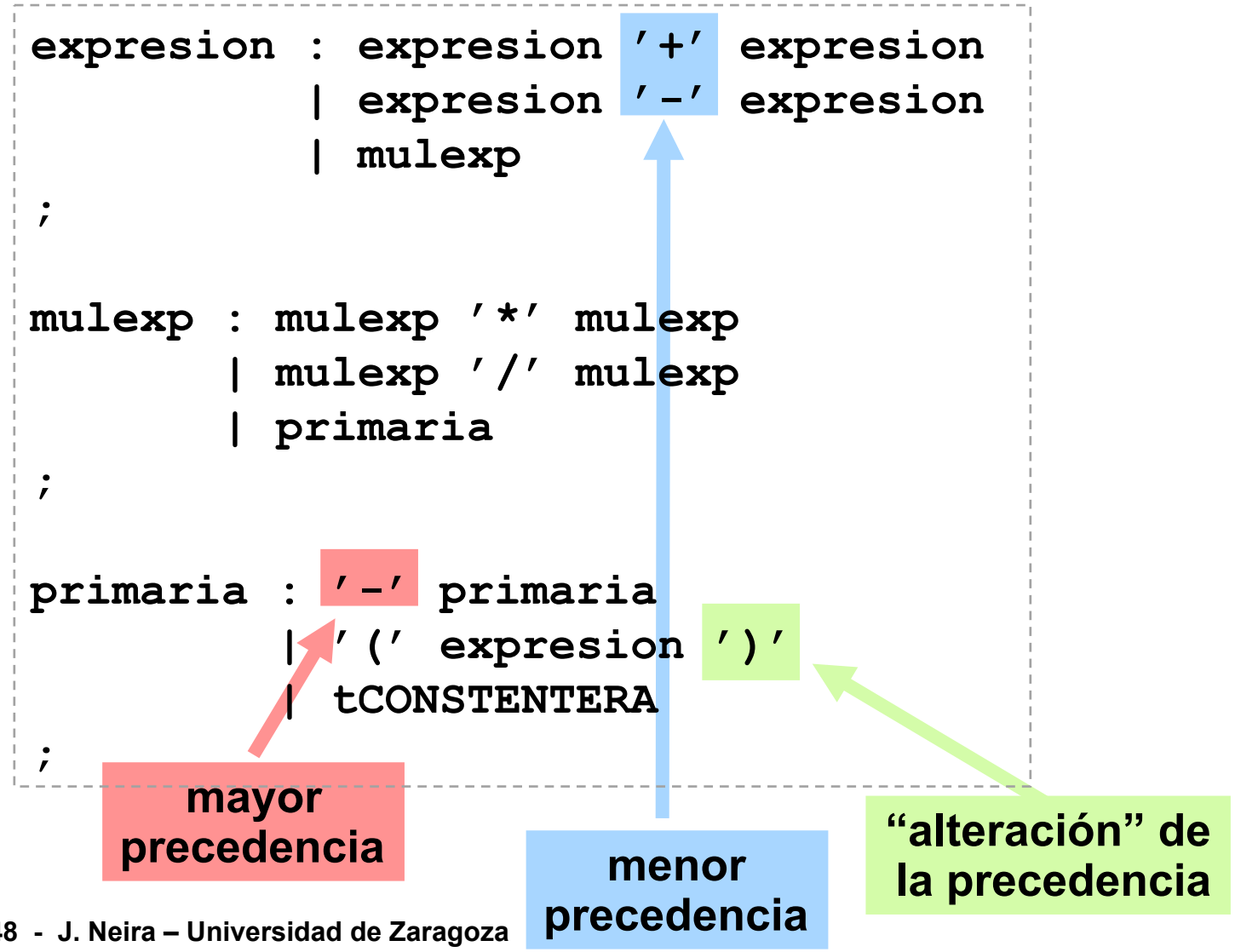
- ¡Es ambigua!



- No se ha especificado la **precedencia** ni **asociatividad** de los operadores

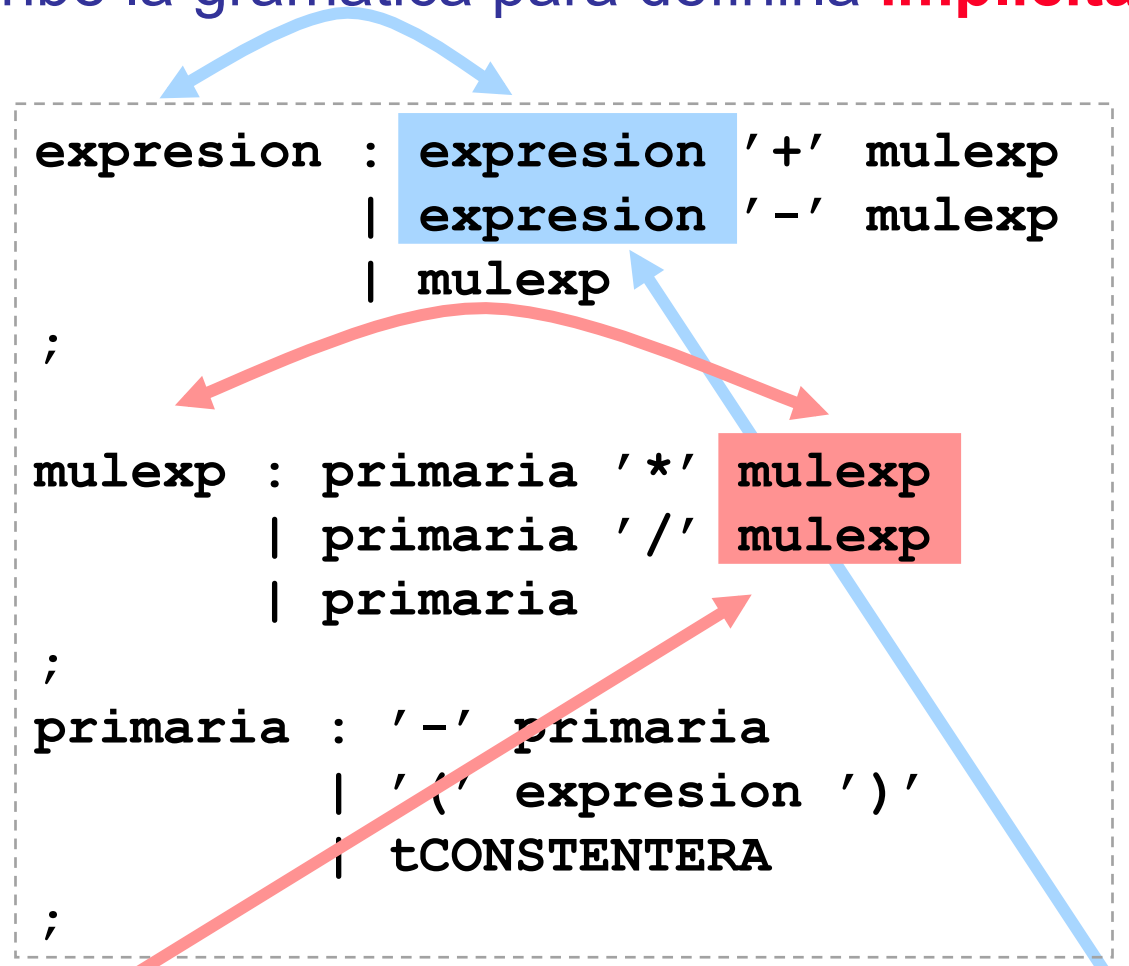
Precedencia: solución implícita

- Se re-escribe la gramática para definirla **implícitamente**



Asociatividad: solución implícita

- Se re-escribe la gramática para definirla **implícitamente**



de derecha a izquierda (¡normalmente es al revés!)

de izquierda a derecha

Precedencia y Asociatividad: solución explícita

- YACC permite definir ambas cosas **explícitamente**

```
%left '+' '-'
%left '*' '/'
...
%nonassoc UMINUS
%%
expr : expr '+' expr
       ...
     | expr tAND expr
       ...
     | '(' expr ')'
     | '-' expr %prec UMINUS
     | tCONSTENTERA
       ...
;
```

Precedencia: de menor a mayor

- `%left` indica la asociatividad de los operadores (también existe `%right`).
- El orden de las declaraciones determina la precedencia de los operadores.
- A cada producción se le asocia la precedencia del *token* ***más a la derecha.***
- Los conflictos *shift/reduce* se resuelven utilizando la precedencia.

```
expr '+' expr '*'
```

```
¿preced('*') > preced('+')?
•si, shift '*'
•no, reduce expr '+' expr -> expr
```

- Parser pequeño y eficiente.

Precedencia

`2*3+4`

- Si el * tuviese mayor precedencia que el +:

```
Reading (2)
Shifting (tCONSTENTERA)
Reducing tCONSTENTERA  -> expr
Reading ('*')
Shifting ('*')
Reading (3)
Shifting (tCONSTENTERA)
Reducing tCONSTENTERA  -> expr
Reducing expr '*' expr -> expr
Reading ('+')
Shifting ('+')
Reading (4)
Shifting (tCONSTENTERA)
Reducing tCONSTENTERA  -> expr
Reducing expr '+' expr -> expr
```

`10`

- Si el + tuviese mayor precedencia que el *:

```
Reading (2)
Shifting (tCONSTENTERA)
Reducing tCONSTENTERA  -> expr
Reading ('*')
Shifting ('*')
Reading (3)
Shifting (tCONSTENTERA)
Reducing tCONSTENTERA  -> expr
Reading ('+')
Shifting ('+')
Reading (4)
Shifting (tCONSTENTERA)
Reducing tCONSTENTERA  -> expr
Reducing expr '+' expr -> expr
Reducing expr '*' expr -> expr
```

`14`

Asociatividad

2-3+4

- Asociatividad por la izquierda:

```
Reading (2)
Shifting (tCONSTENTERA)
Reducing tCONSTENTERA  -> expr
Reading ('-')
Shifting ('-')
Reading (3)
Shifting (tCONSTENTERA)
Reducing tCONSTENTERA  -> expr
Reducing expr '-' expr -> expr
Reading ('+')
Shifting ('+')
Reading (4)
Shifting (tCONSTENTERA)
Reducing tCONSTENTERA  -> expr
Reducing expr '+' expr -> expr
```

3

- Asociatividad por la derecha:

```
Reading (2)
Shifting (tCONSTENTERA)
Reducing tCONSTENTERA  -> expr
Reading ('-')
Shifting ('-')
Reading (3)
Shifting (tCONSTENTERA)
Reducing tCONSTENTERA  -> expr
Reading ('+')
Shifting ('+')
Reading a (4)
Shifting (tCONSTENTERA)
Reducing tCONSTENTERA  -> expr
Reducing expr '+' expr -> expr
Reducing expr '-' expr -> expr
```

-5

Asignación

```
i := j + 42;
```

```
%union{
  struct { char     *nombre;
           SIMBOLO *simbolo;
  } identificador;
}
%type                    tIDENTIFICADOR
```

```
asignacion:
      tIDENTIFICADOR
      {
      }
      tOPAS expresion
      {
      }
;
```

lex debe buscarlo en la tabla

Mejor...

- `$$` se refiere al atributo sintetizado, ***sólo en la última acción de la producción.***
- Acciones intermedias: `$$` atributo asociado con la acción.
- Esto permite transmitir información entre acciones

```
asignacion:
        tIDENTIFICADOR
        {
          $<ok>$ = FALSE;
          if ...
          else $<ok>$ = TRUE;
        }
        tOPAS expresion
        {
          if ($<ok>2 &&
               (($4 != DESCONOCIDO)&&
               ($1.simbolo->variable != $4))
               error_semantico ("tipos incompatibles");
          $$ ...
        }
;
```

```
%union{
   ...
   int ok;
   ...
}
```

Asignación a vectores

- Los atributos también pueden utilizarse para contar elementos.

```
entero v[10];
...
v := [10,-5,4,i,j+1];
```

```
%union{
   int cantidad;
}
%type <cantidad> lista_expresiones
```

```
asignacion_vector:
      tIDENTIFICADOR
      {...}
      tOPAS '[' lista_expresiones ']'
      {...
        if ($1.simbolo->dimension < $5)
            error_semantico ("cantidad incorrecta");
      }
;
lista_expresiones :
      expresion
      {$$ = 1;}
|     lista_expresiones ',' expresion
      {$$ = $1 + 1;}
```

2. Estructuras de control

- **Sentencias**: las acciones efectúan las comprobaciones
- No se sintetiza ningún atributo

```
seleccion: tSI expresion
           
           tENT
           lista_instrucciones
           resto_seleccion
           tFSI
;
mientras_que: tMQ expresion
           ...
           lista_instrucciones
           tFMQ
;
```

Estructuras de Control

- Instrucciones tipo **`break`** y **`continue`**

```
a := 10;

mq a > 0
   ...
   si a = 0
      continuar
   fsi
   ...
fmq
```

```
%{
int bucle = 0;
%}

%%
...
continuar :
      {
        if (!bucle)
          error_semantico(
            "Solo dentro de bucles.");
      }
;
mientras_que :
      tMQ expresion
      {
        bucle = TRUE;
      }
      lista_instrucciones
      {
        bucle = FALSE;
      }
      tFMQ
```

¿Problemas?

Estructuras de Control

¡Puede haber bucles anidados!

```
a := 10;

mq a > 0
   ...
   mq b > 0
   ...
   fmq
   si a = 0
      continuar
   fsi
   ...
fmq
```

Mejor contar los bucles:

```
%{
int bucle = 0;
%}
%%
...
continuar :
        {
        if (bucle > 0)
            ;
        else
            error_semantico (
              "Solo dentro de bucles.");
        }
;
mientras_que:
        tMQ expresion
        {++bucle;}
        lista_instrucciones
        {--bucle;}
        tFMQ
;
```

3. Declaración de Variables

- **Declaraciones tipo C:** El tipo está definido al procesar los identificadores.

```
int     i, j, k;
float   x, y, z, w;
```

```
%union{
  int tipo;
}
%type <tipo> tipo_variables
```

```
declaracion : tipo_variables identificadores ';'
;
tipo_variables
     : tENTERO    { $$ = ENTERO;    }
     | tCARACTER  { $$ = CHAR;      }
     | tBOOLEANO  { $$ = BOOLEANO;  }
;
identificadores
     : tIDENTIFICADOR
       {
       }
     | identificadores ',' tIDENTIFICADOR
       { . . . }
```

Atributos heredados

```
declaracion : tipo_variables identificadores ';'
;
tipo_variables
     : tENTERO    { $$ = ENTERO;    }
     | tCARACTER  { $$ = CHAR;      }
     | tBOOLEANO  { $$ = BOOLEANO;  }
;
identificadores
     : tIDENTIFICADOR
       {
       if ($1.simbolo == NULL)
         introducir_variable (tabsim, $1.nombre,
                              $<tipo>0, ¿nivel?, ...);
       else
         error_semanti
       }
       | identificador
       { ... }
```

Atributo del símbolo almacenado en la pila antes de `identificadores`

`$0, $-1,...`

OJO:

```
leer : tLEER '(' identificadores ')'
;
```

Ahora antes de `identificadores` aparece también `'('`.

Bloques

```
%{
int nivel;
%}
...
void abrir_bloque ()
{
  nivel++;
  ...
}
void cerrar_bloque ()
{
  ...
  nivel--;
}
```

```
declaracion_accion:
      cabecera_accion ';'
      declaracion_variables
      declaracion_acciones
      bloque_sentencias
      {
        ...
        cerrar_bloque();
      }
;
cabecera_accion:
      tACCION tIDENTIFICADOR
      {
      ...
      abrir_bloque();
      }
      parametros_formales
;
```

El bloque se abre *después* de procesar el identificador y *antes* de los parámetros

```
identificadores
     : tIDENTIFICADOR
       {
       if ($1.simbolo == NULL) ||
            ($1.simbolo->nivel != nivel))
          introducir_variable (tabsim, $1.nombre,
                               $<tipo>0, nivel, ...);
       else
         error_semantico ("id duplicado.");
       }
```

Declaración de Variables

- **Declaraciones tipo ADA:** El tipo NO está definido al procesar los identificadores.

```
I, J, K : Integer;
X, Y, Z, W : Float;
```

```
%union{
  LISTA lista;
}
%type <lista> identificadores
```

G1

```
declaracion
        : identificadores ':' tipo_variables ';'
          {
            declaracion_variables ($1, $3);
          }
...
;
tipo_variables : ...
;
identificadores : tIDENTIFICADOR
                  { $$ = crear_lista ($1); }
                | identificadores ',' tIDENTIFICADOR
                  { $$ = anadir ($1, $3);  }
```

Es necesario usar listas de ids.

Alternativa

- Modificación de la gramática
- ¡Cambia el orden de inserción!

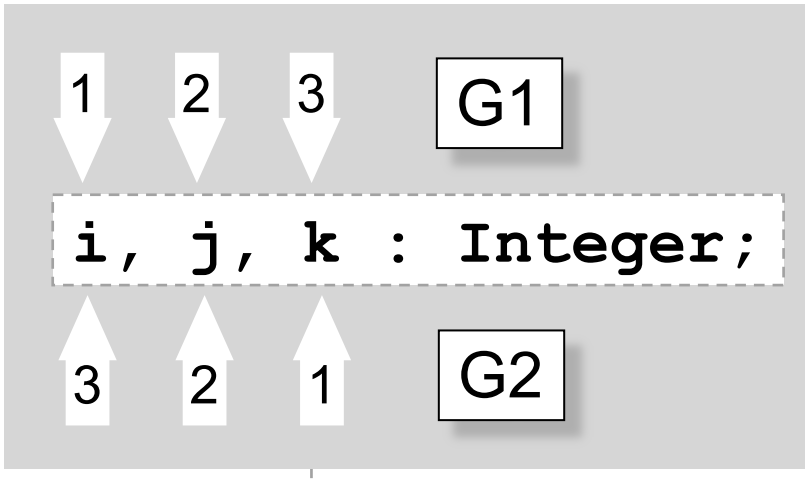
```
%union{
   int tipo;
}
%type <tipo> resto_declaracion
```

```
declaracion : tIDENTIFICADOR
              resto_declaracion
              {...
              }
;
```

G2

```
resto_declaracion : ',' tIDENTIFICADOR
                    resto_declaracion
                  {
                     $$ = $3;...
                  }
                  | ':' tipo_variables
                  {
                     $$ = $2;
                  }
```

Evita el manejo de listas de nombres.



¿es relevante el orden en la declaración?

- En Pascal, el orden NO es relevante

```
var  i, j, k : integer;
     x, y, z, w : real;
```

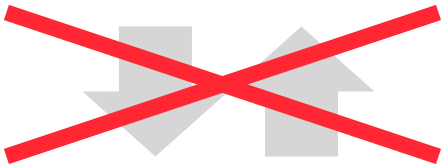


```
var  w, y, z, x : real;
     k, i, j : integer;
```

- NO hay información semántica en el orden de declaración.

- En C, ADA ¡el orden si puede ser relevante!

```
int a = pop(),
    b = pop();
...
push(a-b);
```



```
int b = pop(),
    a = pop();
...
push(a-b);
```

- SI hay información semántica en el orden de declaración (la secuencialidad de los inicializadores).

Ventajas sintácticas

- C:

```
char*  i, j, k;
```

- ADA:

```
i, j, k : access character;
```

4. Procedimientos y Funciones

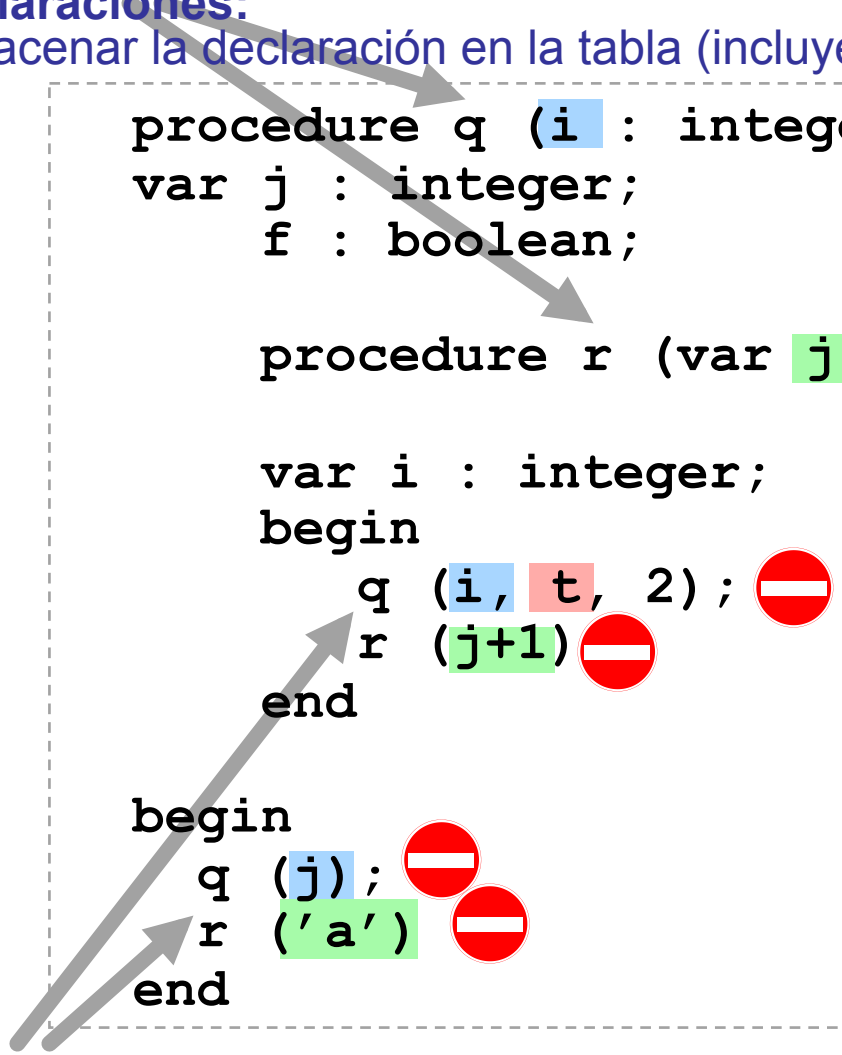
Declaraciones:
Almacenar la declaración en la tabla (incluyendo los parámetros).

```
procedure q (i : integer; var t : boolean);
var j : integer;
    f : boolean;

    procedure r (var j : integer);

    var i : integer;
    begin
       q (i, t, 2);
       r (j+1)
    end

begin
  q (j);
  r ('a')
end
```

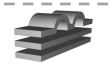


Invocaciones:
Verificar que la cantidad, tipo y clase de los argumentos sea correcta.

Declaraciones

```
%union {....
          struct {SIMBOLO *simbolo; char *nombre;} ident;
          LISTA *lista;
.....}
%type <ident> tIDENTIFICADOR
%type <lista> parametros_formales
%%
accion : tACCION tIDENTIFICADOR
          {
            $<simbolo>$   = NULL;
            if (($2.simbolo == NULL) ||
                ($2.simbolo->nivel != nivel))
              $<simbolo>$ = introducir_accion
                              (tabsim, $2.nombre, ...);
            ....
          }
          parametros_formales
          {
            if ($<simbolo>3 != NULL)
              $<simbolo>3->parametros = $4;
          }
          ';' declaracion_variables
              declaracion_acciones
              bloque_instrucciones
```

```
accion p(
   val entero    i, j;
   ref booleano  k, l, m;
   val caracter  n, o)
```



Declaraciones

- Es necesario almacenar el símbolo correspondiente a cada parámetro, su dirección asignada, y el orden en que ha sido declarado (necesario para verificar invocaciones).

```
%type <lista> parametros lista_parametros
%%
parametros_formales : '(' lista_parametros ')'
                      {
                        $$ = $2;
                      }
                    | {
                        crear_vacia (&($$));
                      }
                    ;

lista_parametros : parametros
                   {
                     $$ = $1;
                   }
                 | lista_parametros ';' parametros
                   {
                     concatenar (&($1), $3);
                     $$ = $1;
                   }
                 ;
```

Acción por defecto

Invocaciones

- Cada argumento debe compararse por orden de aparición con la correspondiente declaración del parámetro.
  - Los tipos deben coincidir
  - Los argumentos a parámetros por referencia sólo pueden ser variables.

```
invocacion_accion : tIDENTIFICADOR
                  | tIDENTIFICADOR '(' argumentos ')'
;

argumentos : expresion
           {
             LISTA pars;
             SIMBOLO *p;

             pars = (*($<identificador.simbolo>-1)).parametros;
             if (longitud_lista (pars) < 1)
                error ("Accion no tiene parametros.");

             p = observar (pars, 1);
             if ((*p).tipo != $1.tipo)
                ..... /*verificaciones */

             $$ = 1;
           }
           | argumentos ',' expresion
             ....... /* algo parecido para $1 + 1 */
```

5. Perspectiva

- Sobrecarga de Operadores

```
function "/" (i, j : integer) return rational is ...

Put(i/j);
```

- Interpretaciones posibles de '/':

```
integer / integer ->  integer
integer / integer ->  rational
```

- Acción semántica:

```
expresion : expresion '/' expresion
            {
              if (($1.tipo == ENTERO) &&
                  ($3.tipo == ENTERO))
                 ??????????????????
            }
  ;
```

¡hay dos posibilidades!

5. Perspectiva

- **Paso de parámetros:**
  el orden posicional y parámetros por valor/referencia es UNO de los posibles mecanismos de paso de parámetros.

Orden nominal

```
procedure Quadriatic (A, B, C : in float ......
...
Quadratic(B=>M, C=>N, A=>L, .....);
```

orden diferente

Valores por defecto

```
procedure Quadriatic (A : in float := 0....
...
Quadratic (B=>M, C=>N, .....);
```

A no aparece

El lenguaje se enriquece pero la complejidad del compilador aumenta.