Lección 6: Introducción al análisis semántico

- Introducción al análisis semántico
- Gramáticas atribuídas
- Evaluación de atributos
- Evaluación de atributos. Recorrido de árboles
- Evaluación de atributos. Evaluación "al vuelo"
- Tratamiento de atributos en Yacc
- Evaluación ascendente de atributos sintetizados
- Evaluación ascendente de atributos heredados
- Ejemplos

Introducción al análisis semántico

- El objetivo final de un compilador es la generación de código
  - esto requiere el tratamiento de la información semántica de los símbolos
- ¿Por qué "semántica"?
  - la fase de síntesis comienza con la generación de código (intermedio u objeto)
  - el código generado es lo que da el "significado" a la estructura sintáctica reconocida
- Procesamiento semántico:
  - chequeo semántico
  - generación de código

- Propuestas por Knuth (1968)
  - incluir información semántica en la GLC del lenguaje
- Cada símbolo gramatical tiene asociados un conjunto de **atributos**
  - representan información semántica
    - » Ejemplos:
      - para una constante: **`valor, tipo`**
      - para un identificador: **`string, tipo, valor actual, posición en memoria`**

`Gramática atribuida`

`Gramática en la que cada producción tiene asociado un conjunto de reglas para los valores de los atributos`

- Uso:
  - especificar y calcular atributos
  - generar código

- Ejemplo:

Atributos:
S,E,T,F,ID: (val: Entero)

$S \rightarrow E$	S.val:=E.val
$E \rightarrow E$ **OR** T	E.val:=E_1.val $\vee$ T.val
$E \rightarrow T$	E.val:=T.val
$T \rightarrow T$ **AND** F	T.val:=T_1.val $\wedge$ F.val
$T \rightarrow F$	T.val:=F.val
$F \rightarrow (E)$	F.val:=E.val
$F \rightarrow$ **NOT** E	F.val:=$\neg$E.val
$F \rightarrow$ **ID**	F.val:=ID.val
$F \rightarrow$ **TRUE**	F.val:=1
$F \rightarrow$ **FALSE**	F.val:=0

producciones

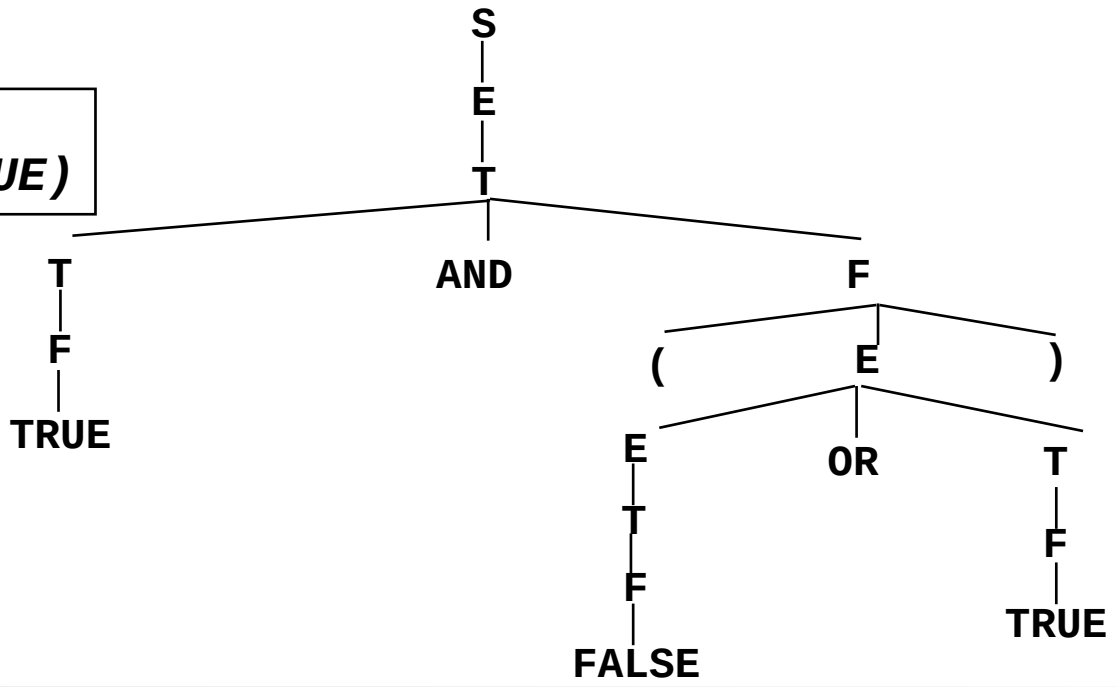
reglas semánticas

- También el árbol de sintaxis que construya el analizador sintáctico deberá incluir la evaluación de los atributos

árbol de sintaxis decorado

- Ejemplo:

TRUE AND (FALSE OR TRUE)

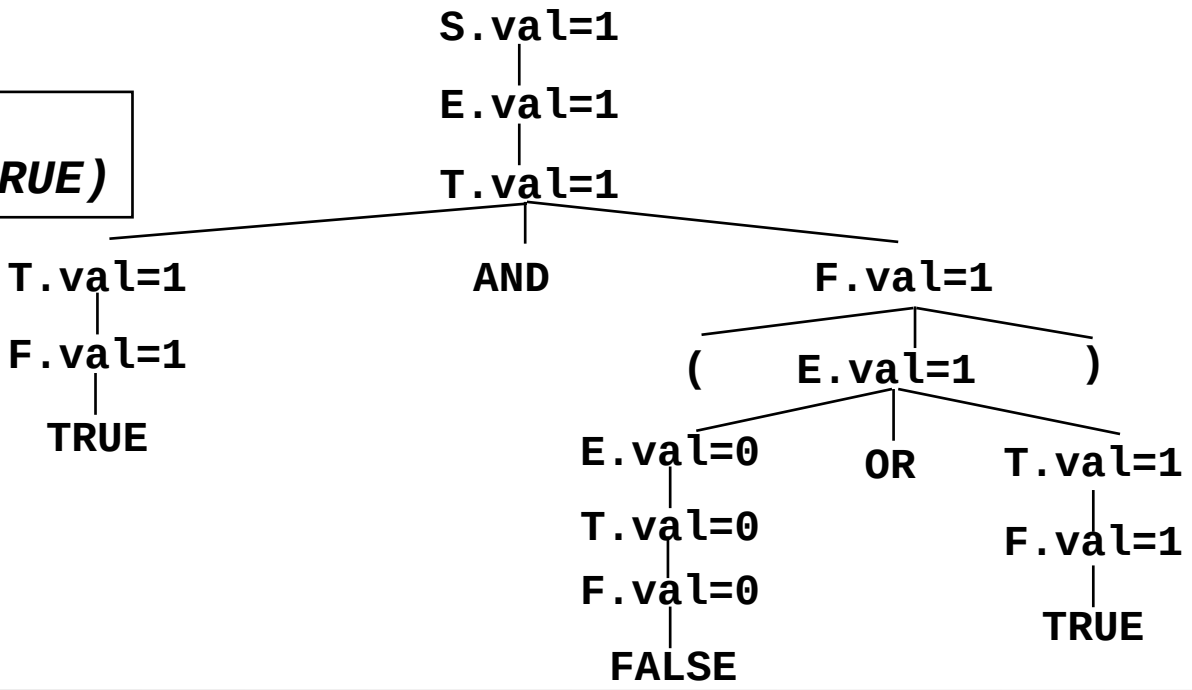


- También el árbol de sintaxis que construya el analizador sintáctico deberá incluir la evaluación de los atributos

árbol de sintaxis decorado

- Ejemplo:

TRUE AND (FALSE OR TRUE)

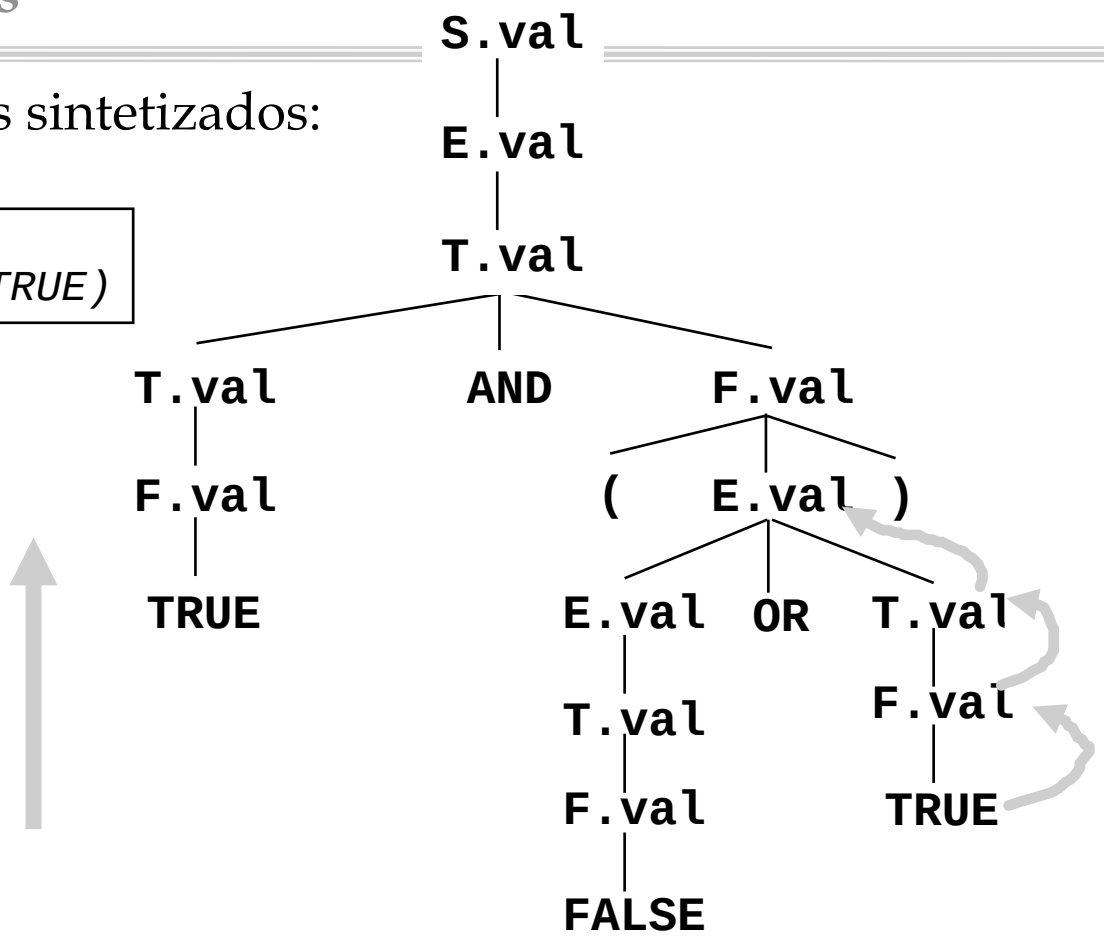


- Dos tipos de atributos
  - Atributos **sintetizados**:
    - » se obtienen a partir de atributos de los descendientes en el árbol
    - » pasan información "hacia arriba"
    - » por lo tanto: fáciles de manejar para analizadores bottom-up
  - Atributos **heredados**:
    - » su valor depende de atributos del padre y/o de los hermanos
    - » especialmente útiles para expresar propiedades dependientes del contexto
      - alcance de una variable
      - tipo de una variable
    - » fáciles de obtener en analizadores top-down

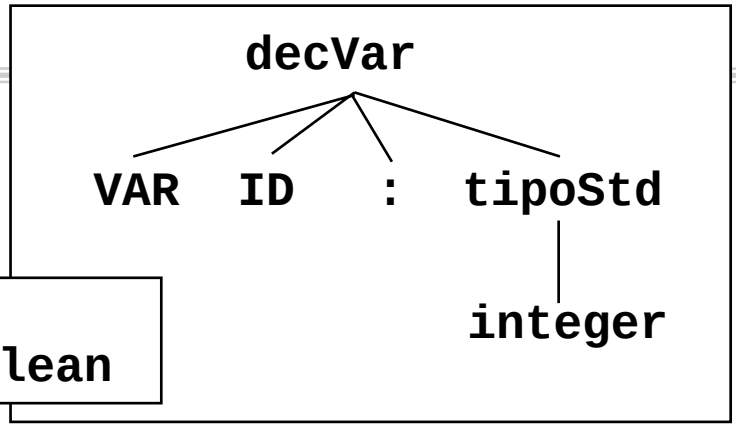
- Ejemplo de atributos sintetizados:

```
TRUE AND
       (FALSE OR TRUE)
```

$S \rightarrow E$;
$E \rightarrow E$ **OR** T
$E \rightarrow T$
$T \rightarrow T$ **AND** F
$T \rightarrow F$
$F \rightarrow (E)$
$F \rightarrow$ **NOT** E
$F \rightarrow$ **ID**
$F \rightarrow$ **TRUE**
$F \rightarrow$ **FALSE**

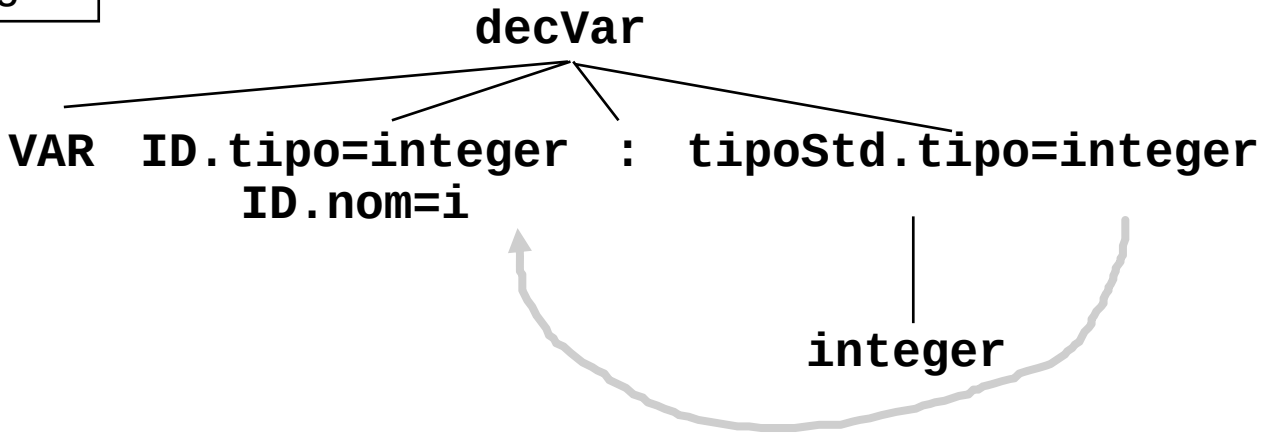


- Ejemplo de atributos heredados




```
decVar → VAR ID : tipoStd
tipoStd → integer | real | boolean
```

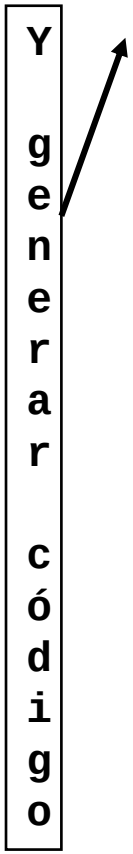
```
VAR i:integer
```



Gramáticas atribuídas

- Sobre atributos en terminales:
  - sus atributos sintetizados son suministrados por el analizador léxico
    - » valor de una constante
    - » nombre de un identificador
    - » ...
- Sobre atributos de no terminales:
  - pueden tener tanto heredados como sintetizados
  - respecto al símbolo inicial
    - » se asume que no tiene heredados
    - » alternativamente, puede tenerlos, pero sus valores se suministran como parámetros en la invocación al tratamiento semántico

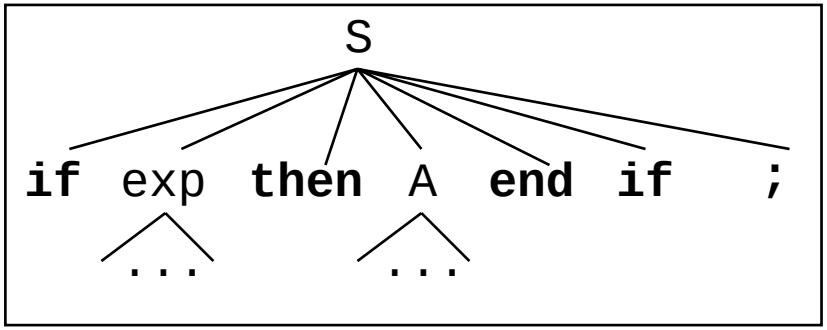
Evaluación de atributos



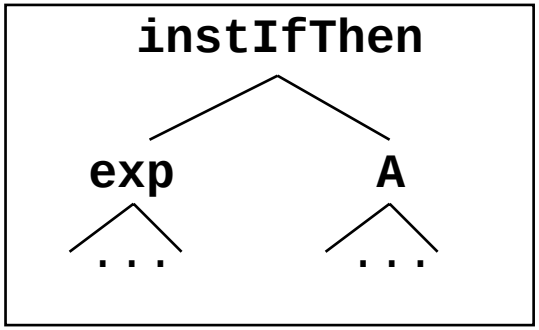
- Dos formas de evaluar los atributos:
  - crear y recorrer un árbol de sintaxis
    - » construir un árbol
    - » realizar varias pasadas sobre él
    - » mayor modularidad
    - » compilador de varias pasadas
  - "al vuelo" ("on the fly")
    - » los atributos se evalúan conforme se realiza el análisis sintáctico
    - » compiladores de una pasada
    - » menos requerimiento de memoria
    - » más restricciones en la organización de la evaluación de atributos

- Construcción de un árbol: **árbol de sintaxis abstracta**
- Ejemplo: $S \rightarrow$ ***if*** *exp* ***then*** *A* ***end if*** ***;***

árbol de sintaxis ***concreta***



árbol de sintaxis ***abstracta***



- Si bien para la construcción se visitará el primero, para el análisis semántico se almacenará y pasará el segundo

- Evaluación de izquierda a derecha:
  - se dispone de un árbol con
    - » atributos heredados del símbolo inicial
    - » atributos sintetizados de las hojas
  - **proceso**: repetidos paseos por el árbol hasta tener todos los atributos calculados
  - **estrategia de recorrido**: en profundidad y de izda. a dcha.

Evaluación de atributos. Recorrido de árboles

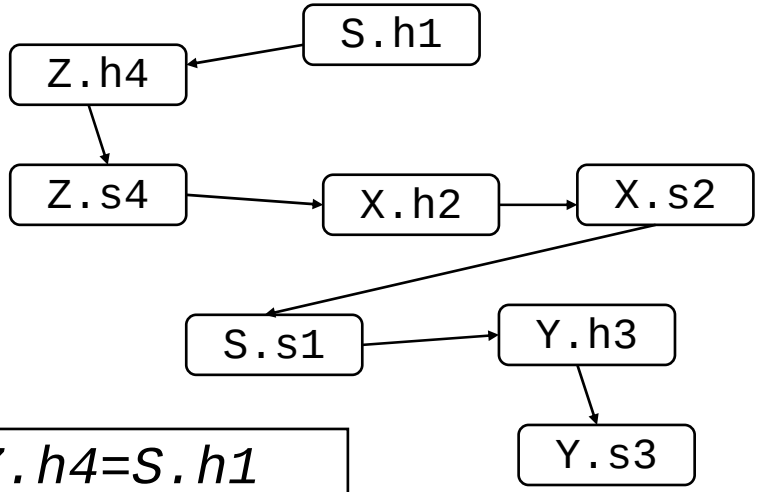
```
Alg evaluaAtrib(ES A:arb. de sintaxis)
Pre: A es un arb. de sintaxis sin ref. "circulares en sus
     atrib." t.q. han sido evaluados: 1)heredados del
     inicial 2)sintetizados de terminales
Post: cada atributo ha sido evaluado
Principio
  Mq queden atributos sin evaluar
     visitarNodo(A,S) /*simb. inicial*/
  FMq
Fin
```

```
Alg visitarNodo(ES A:arb. de sintaxis;E N:nodo)
Principio
  Si N es no terminal /*N → X1...Xm*/
    Para i de 1 a m  /*izda. a dcha.*/
      Si Xi es no terminal
        evaluar atr. her. de Xi que se pueda
        visitarNodo(A,Xi) /*en profundidad*/
      FSi
    FPara
  Fsi
  evaluar atr. sint. de N que se pueda
Fin
```

- Ejemplo

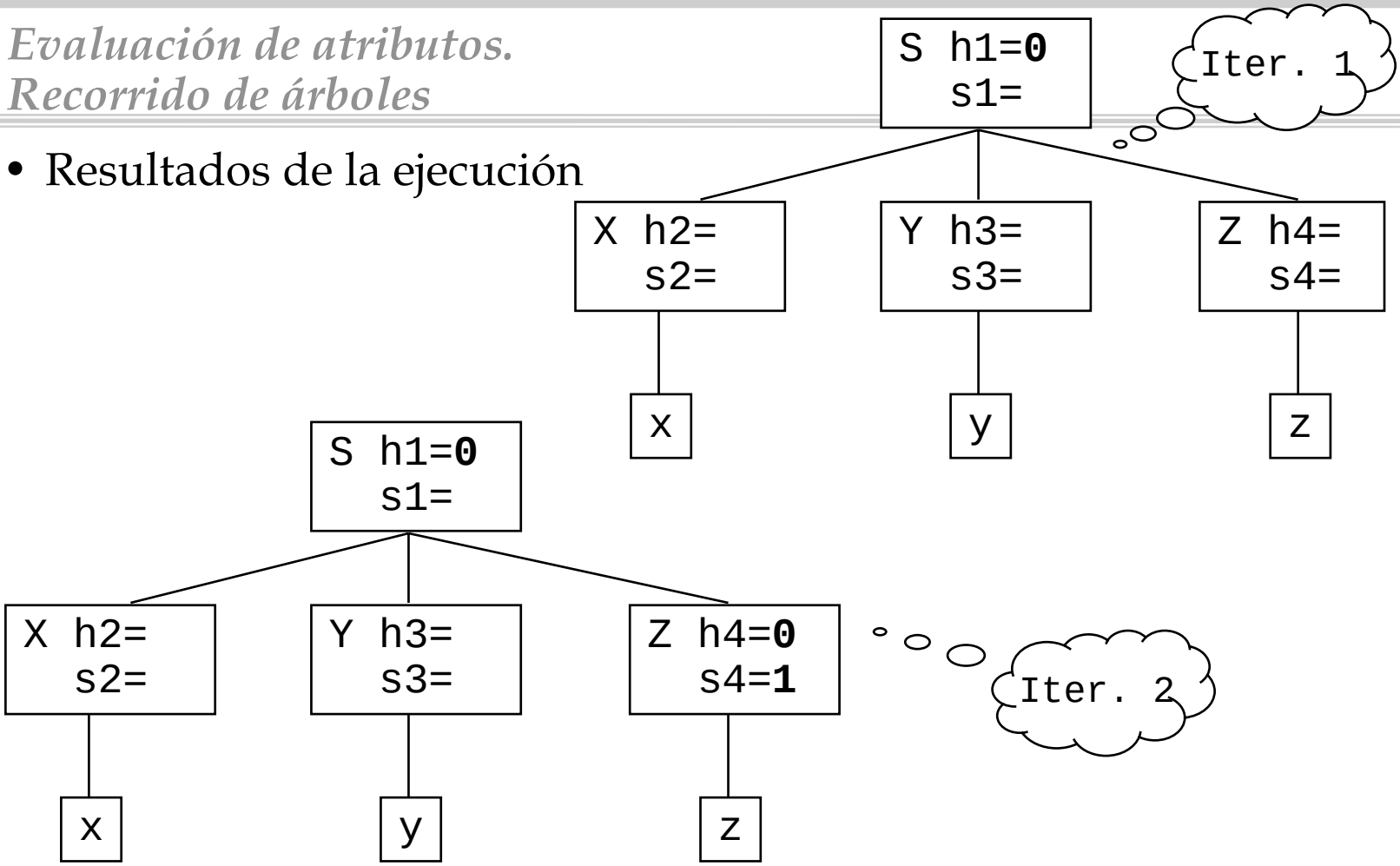
atributos

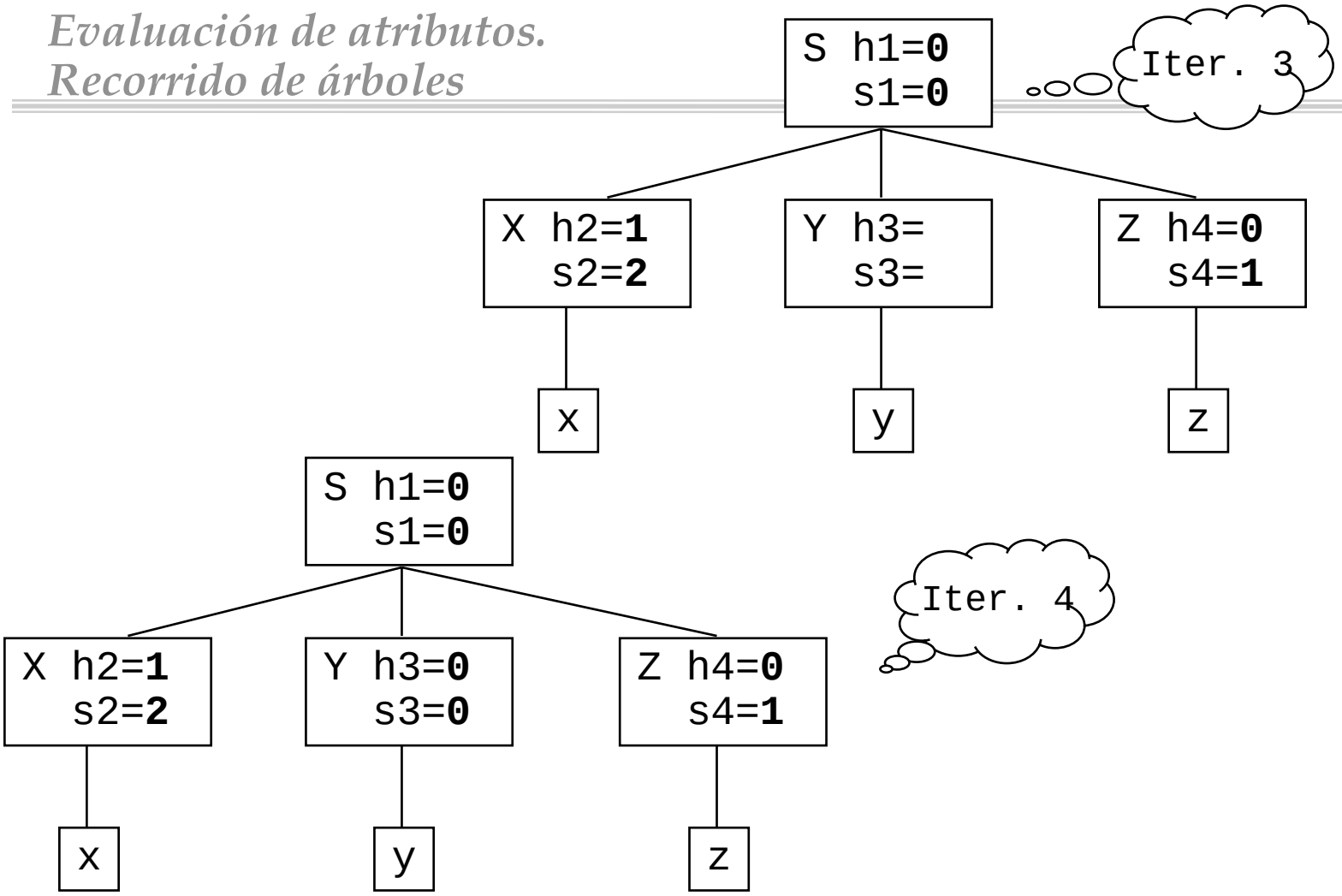
	her.	*sin.*
S	*h1*	*s1*
X	*h2*	*s2*
Y	*h3*	*s3*
Z	*h4*	*s4*



$S \rightarrow X\ Y\ Z$	$Z.h4=S.h1$
	$X.h2=Z.s4$
	$S.s1=X.s2-2$
	$Y.h3=S.s1$
$X \rightarrow \mathbf{x}$	$X.s2=2*X.h2$
$Y \rightarrow \mathbf{y}$	$Y.s3=3*Y.h3$
$Z \rightarrow \mathbf{z}$	$Z.s4=Z.h4+1$

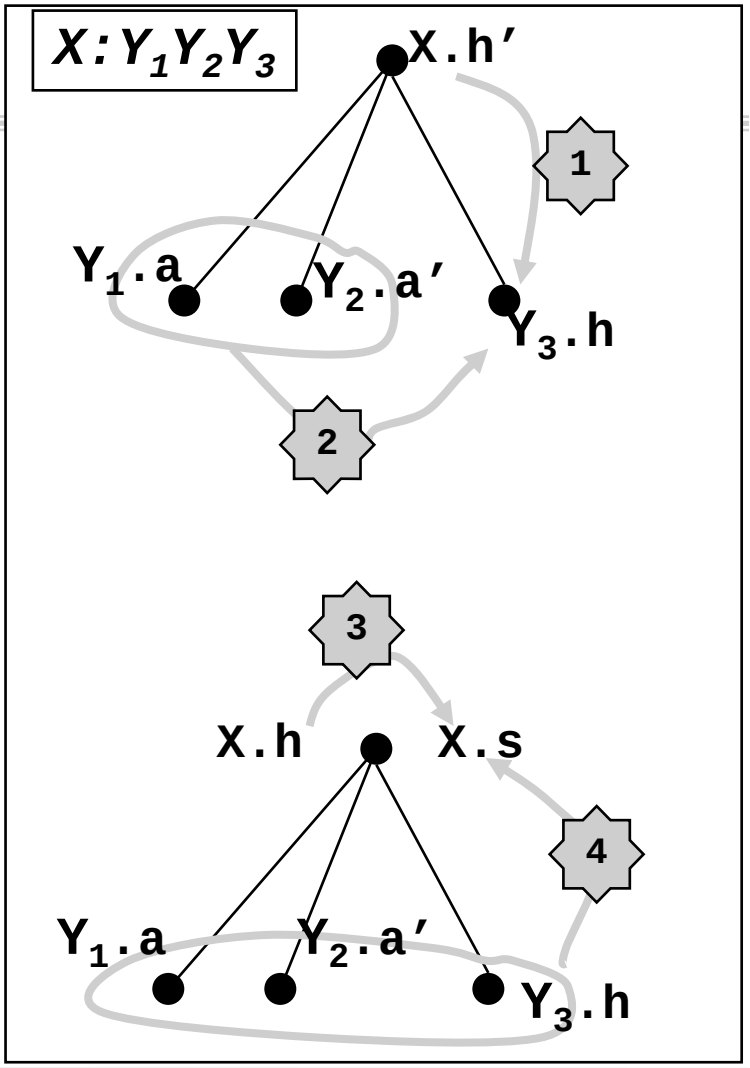
- Resultados de la ejecución




S h1=0
s1=0
Iter. 3
X h2=1
s2=2
Y h3=
s3=
Z h4=0
s4=1
x
y
z
S h1=0
s1=0
Iter. 4
X h2=1
s2=2
Y h3=0
s3=0
Z h4=0
s4=1
x
y
z

- Clase especial: gramáticas "**L-atribuídas**"
  - heredado en parte dcha. dependen de:
    - 1- atributos heredados de parte izda. de la producción
    - 2- atributos a "la izda. dentro de la dcha. de la producción"
  - sintetizados en parte izda. dependen de:
    - 3- sus heredados
    - 4- atributos de la parte dcha.
  - Propiedad: los atributos se evalúan en una pasada

ideales para LL(1)



- Para el método anterior
  - la diferencia entre top-down y bottom-up se establece en la construcción del árbol
  - el recorrido se hace a posteriori, de manera independiente a la forma de anál. sint.
- Evaluación "al vuelo":
  - no hay construcción de árbol entre el sintáctico y el semántico: se realizan a la vez
  - buen modelo para compiladores de una pasada
  - el tratamiento top-down será distinto al bottom-up
- Nos vamos a centrar en el caso de analizadores bottom-up

- Atributos en Yacc

```
S.val:= E.val
E.val:=E1.val ∨ T.val
E.val:=T.val
T.val:=T1.val ∧ F.val
T.val:=F.val
F.val:=E.val
F.val:=¬E.val
F.val:=ID.val
F.val:=true
F.val:=false
```

```
S → E ;
E → E OR T
E → T
T → T AND F
T → F
F → (E)
F → NOT E
F → ID
F → TRUE
F → FALSE
```

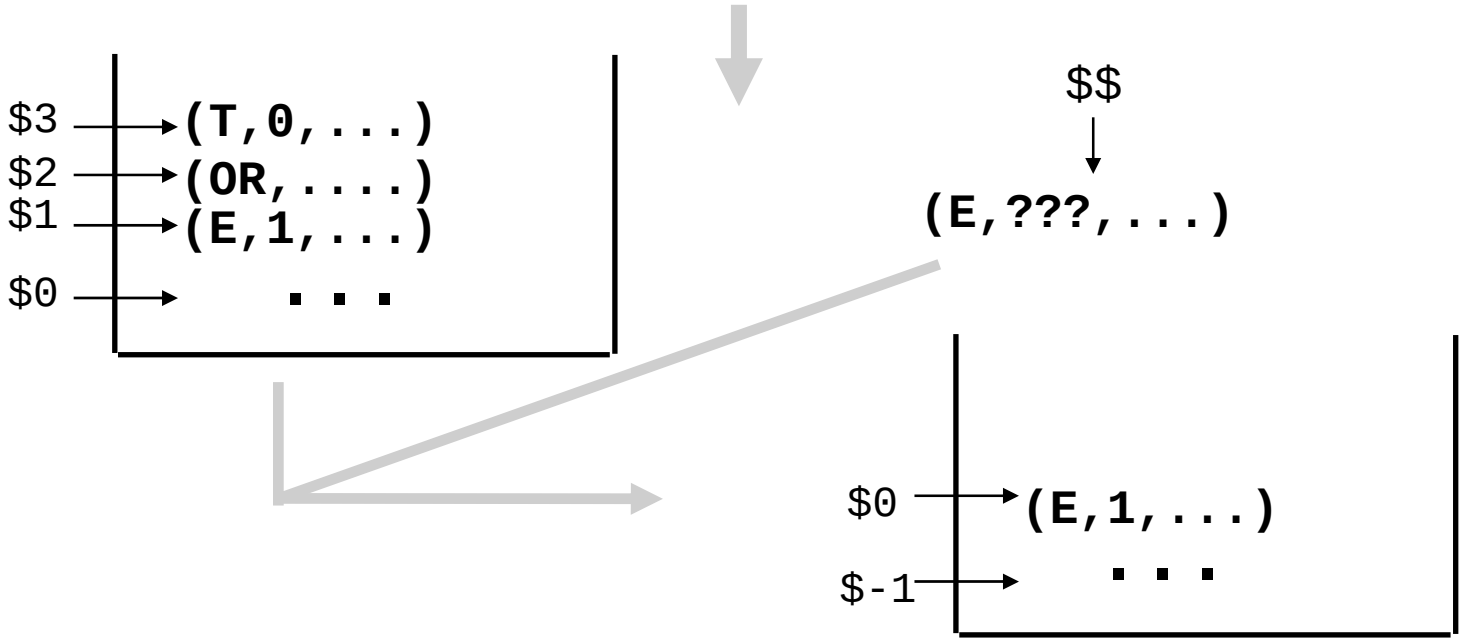
```
S : E ';'       {$$.val=$1.val;};
E : E OR T      {$$.val=$1.val || $3.val;};
E : T           {$$.val=$1.val;};
T : T AND F     {$$.val=$1.val && $3.val;};
T : F           {$$.val=$1.val;};
F : '(' E ')'   {$$.val=$2.val;};
F : NOT E       {$$.val=!$2.val;};
F : ID          {$$.val=$1.val;};
F : TRUE        {$$.val=1;};
F : FALSE       {$$.val=0;};
```

Yacc también admite acciones

- Exactamente, ¿Cómo funciona?

Se evalúa justo antes de la reducción

```
....
E : E OR T  {$$.val=$1.val || $3.val;} ;
....
```



Tratamiento de atributos en Yacc

```
enum CLASEID {ESCALAR,VECTOR};
enum TIPOSBASE {tpINT,tpREAL,tpBOOL};
```

```
%token ID
%type <infoID> ID
%type <exprFactTerm> expr
```

```
instAs: ID tkOPAS expr;
```

```
%union{
    ...
  struct{
     enum TIPOSBASE elTipo;
     union{
        float valReal;
        int valIntBool;
     }valor;
  }exprFactTerm;
  struct{
     enum CLASEID laClase;
     char nombre[30];
     enum TIPOSBASE elTipo;
     union{
        float valReal;
        int valIntBool;
        struct {
          int comInd,finInd;
          void *ptPrimVal;
        } valArray;
     }valor;
  }infoID;
}
```

- El fichero "y.tab.h" contendrá algo como:

```
. . .
typedef union{
  una copia del union declarado
}YYSTYPE;

YYSTYPE yylval;
```

- Por lo que desde LEX se le asigna el nombre del ID como sigue:

```
ID      {letra}({letra}|{digito}|"_")*
. . .
%%
. . .
{ID}    {strcpy(yylval.infoID.nombre,yytext);
         return(ID);}
. . .
```

Tratamiento de atributos en Yacc

- Ejemplo de uso de los atributos Yacc:

```
instAs: ID tkOPAS expr
 { simbolo *pS;
   pS=buscarSimbolo(laTabla,$1.nombre);
   if(pS==NULL)
      errorSem("ID no declarado");
   if(pS->laClase != ESCALAR)
      errorSem("ID debe ser escalar");
   if($1.elTipo != $3.elTipo)
      errorSem("Tipos incompatibles");
   else
     "asignar a
        $1.valor.valReal ó
        $1.valor.valIntBool
         según $3"
 }
;
```

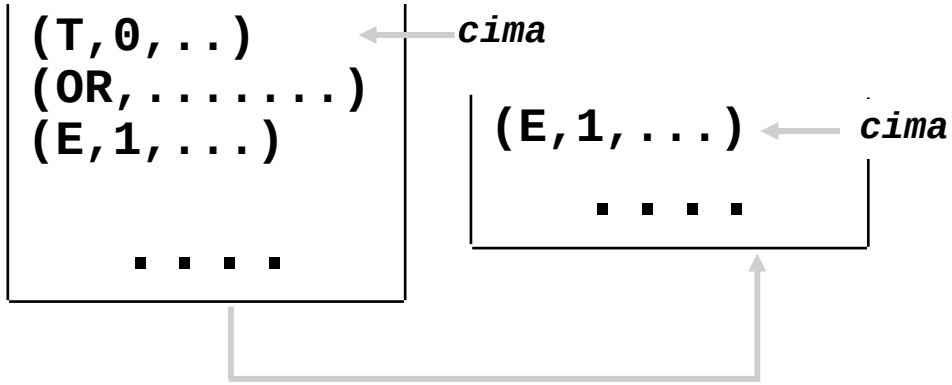
  - Además, uso en relación a `'yylval'`

- Dos tipos de evaluaciones distintas:
  - evaluación de atributos sintetizados
  - evaluación de atributos heredados
- ¿Cómo sintetizar atributos con un analizador LR?
  - almacenar en la pila del analizador los atributos ya sintetizados
  - sintetizar los nuevos justo antes de llevar a cabo la reducción

Evaluación ascendente de atributos sintetizados

- Ejemplo: considerar

$S \rightarrow E$;	$S.val = E.val$;
$E \rightarrow E$ **OR** T	$E_1.val = E_2.val$ **OR** $T.val$
$E \rightarrow T$	$E.val = T.val$
$T \rightarrow T$ **AND** F	$T_1.val = T_2.val$ **AND** $F.val$
$T \rightarrow F$	$T.val = F.val$
$F \rightarrow (E)$	$F.val = E.val$
$F \rightarrow$ **NOT** E	$F.val =$ **NOT** $E.val$
$F \rightarrow$ **ID**	$F.val =$ **ID**$.val$
$F \rightarrow$ **TRUE**	$F.val =$ **TRUE**
$F \rightarrow$ **FALSE**	$F.val =$ **FALSE**



- Luego, evaluar atributos sintetizados es "sencillo"

- Teoría y práctica:
  - la teoría exige que la evaluación de atributos se realice "al final", justo antes de la reducción
  - pero es muy útil realizar acciones/evaluar atributos intercaladas
    - » Ejemplo: traductor a C (p.e.)

```
E : T '+' {printf(" + ");} F ;
```

- Solución: introducción de "marcadores"

```
E : T '+' MARC F ;

MARC : ε {printf(" + ");}
```

Realmente, es lo que hace Yacc

La adición de acciones intercaladas (o "marcadores") puede generar conflictos no existentes en la gramática inicial

IMPORTANTE

- Ejemplo:

```
tontería: uno
        | otro
;
uno:  'A' 'B' 'C' 'D'
;
otro: 'A' 'B' 'C' 'E'
;
```

¡Estupendo!

```
tontería: uno
        | otro
;
uno:  'A' 'B' {printf("uno");} 'C' 'D'
;
otro: 'A' 'B' 'C' 'E'
;
```

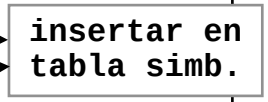
one rule never reduces
conflicts: 1 shift/reduce

¡Conflicto!

Evaluación ascendente de atributos heredados

- Uso de la pila para la evaluación de atributos heredados

```
          .......
enum clases {global,local};
enum tipos  {real, entero};
%}
%union{
    char nombre[30];
    enum clases laClase;
    enum tipos elTipo;
};
%token REAL INTEGER LOCAL GLOBAL ID
%type <laClase> clase
%type <elTipo> tipo
%type <nombre> ID
%%
decVar : clase tipo listaID
;
clase : LOCAL          {$$=local;}
      | GLOBAL         {$$=global;}
;
tipo : REAL            {$$=real;}
     | INTEGER         {$$=entero;}
listaID : ID
        | listaID ID
;
```



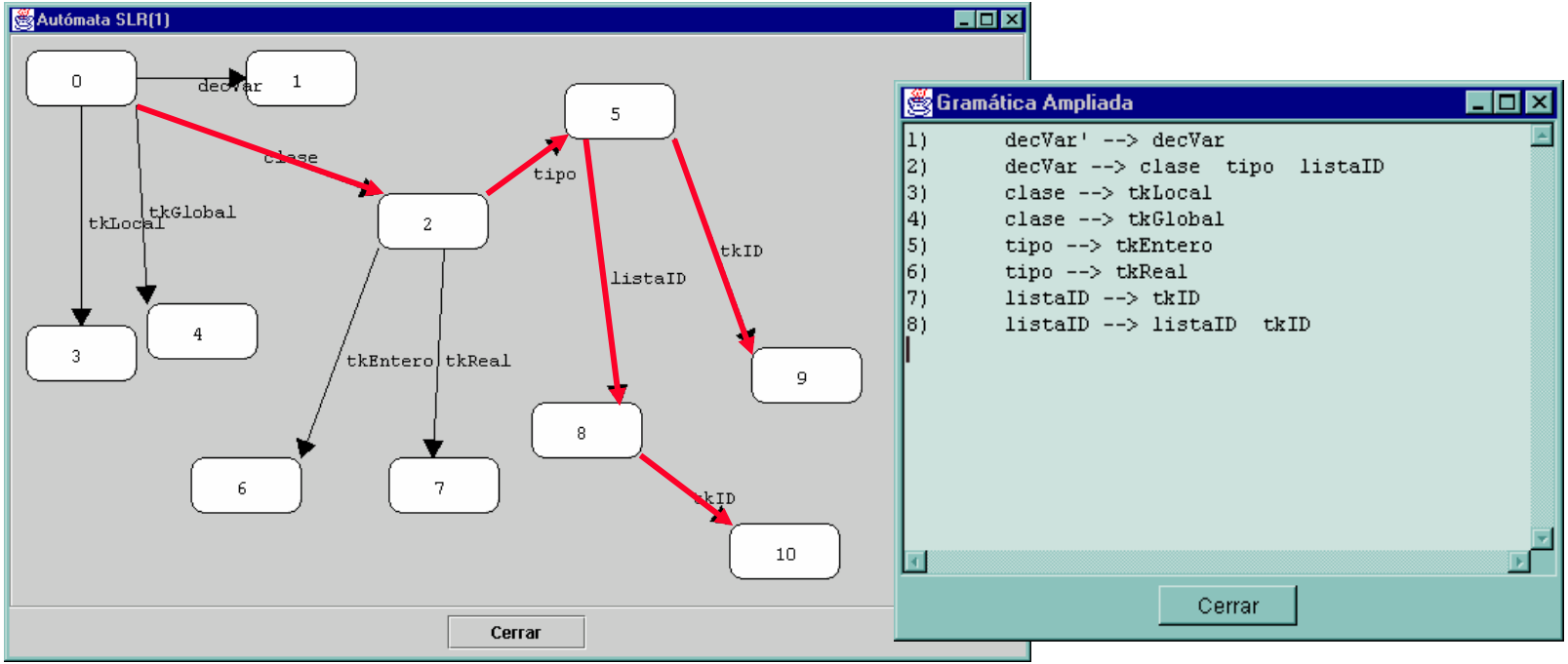
Evaluación ascendente de atributos heredados

- Para insertar en la tabla de símbolos:
- La información está en la pila

```
void IT(char *id,enum tipos t,enum clases c)
```

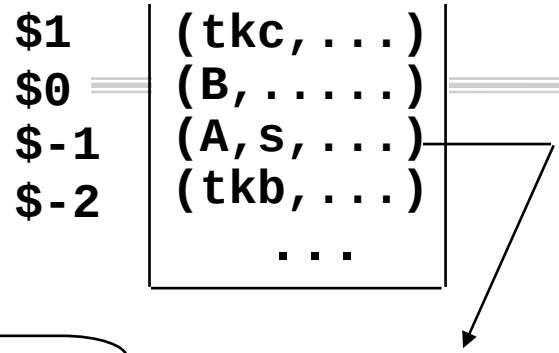
```
.........
%%
decVar : clase tipo listaID
;
clase : LOCAL          {$$=local;}
      | GLOBAL         {$$=global;}
;
tipo : REAL            {$$=real;}
     | INTEGER         {$$=entero;}
;
listaID : ID           {IT($1,$0,$-1);}
        | listaID ID   {IT($2,$0,$-1);}
```

Evaluación ascendente de atributos heredados



Evaluación ascendente de atributos heredados

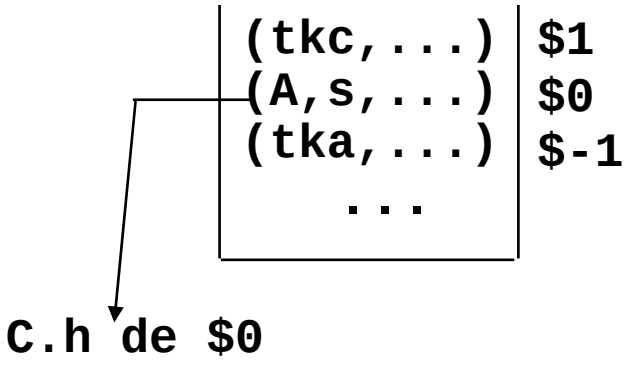
- Pero ojo: puede que la información necesaria para un atr. heredado se encuentre, según la producción, en distintas posiciones de la pila



h: heredado
s: sintetizado

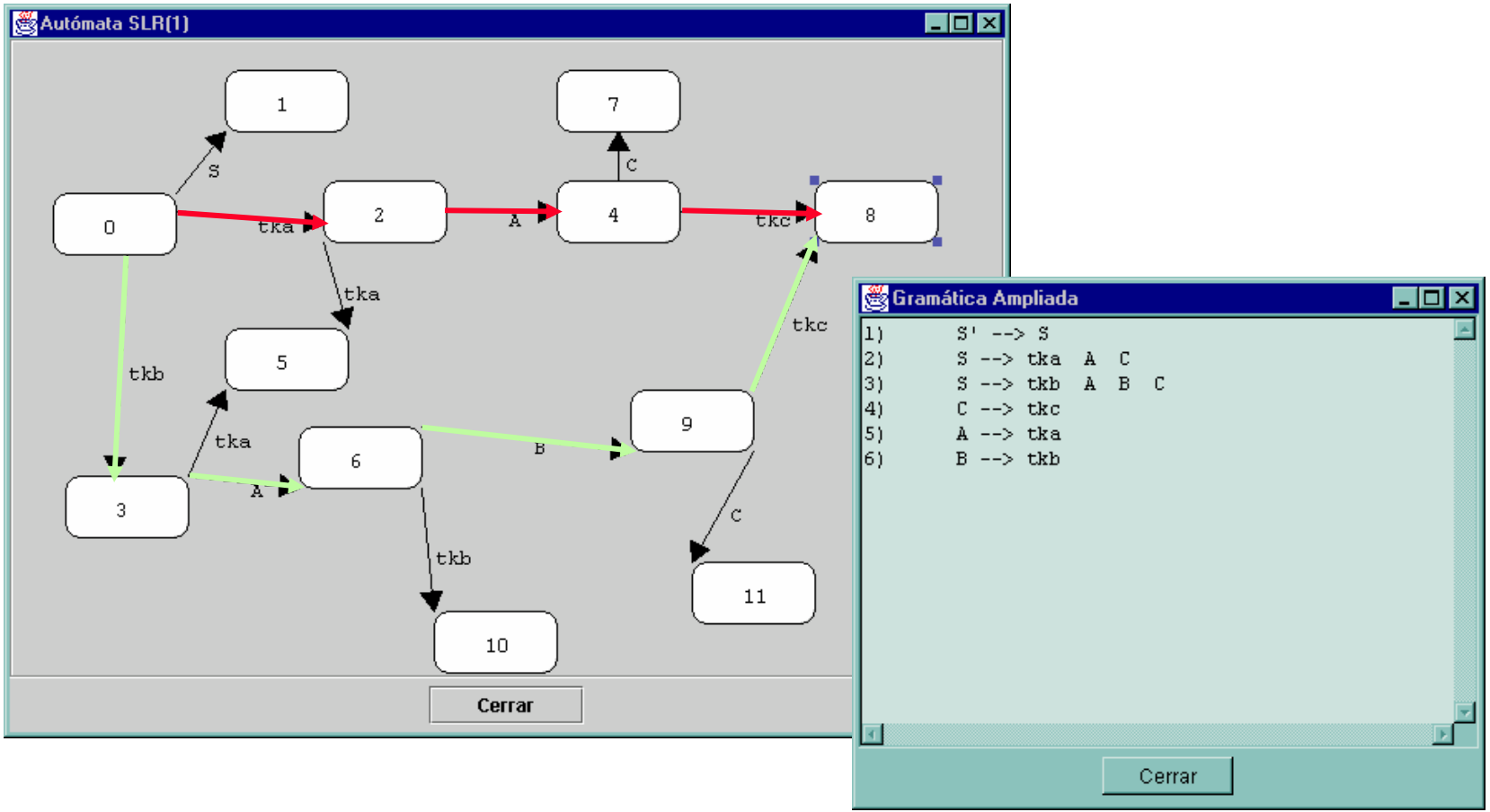
```
S: tka A C      {C.h=A.s}
S: tkb A B C    {C.h=A.s}
C: tkc          {C.s=f(C.h)}
A: tka          {A.s=...}
B: tkb          {B.s=...}
```

```
$3.h = $2.s;
```



- Dos situaciones distintas al reducir `C: tkc`

Evaluación ascendente de atributos heredados

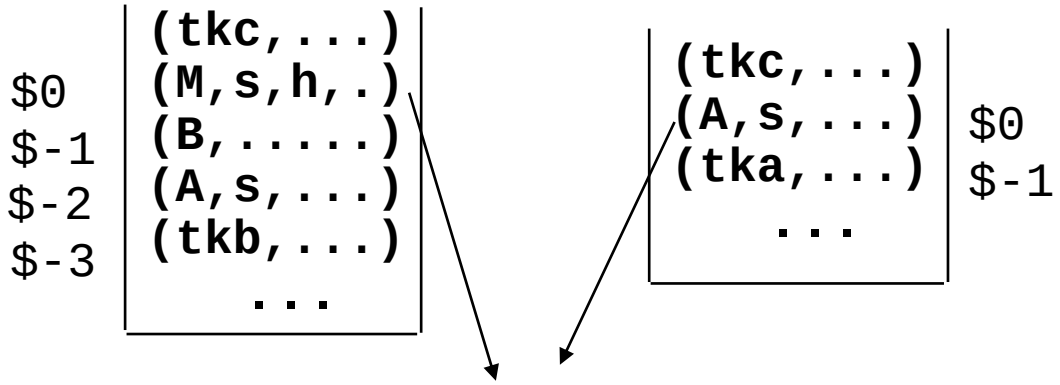


Evaluación ascendente de atributos heredados

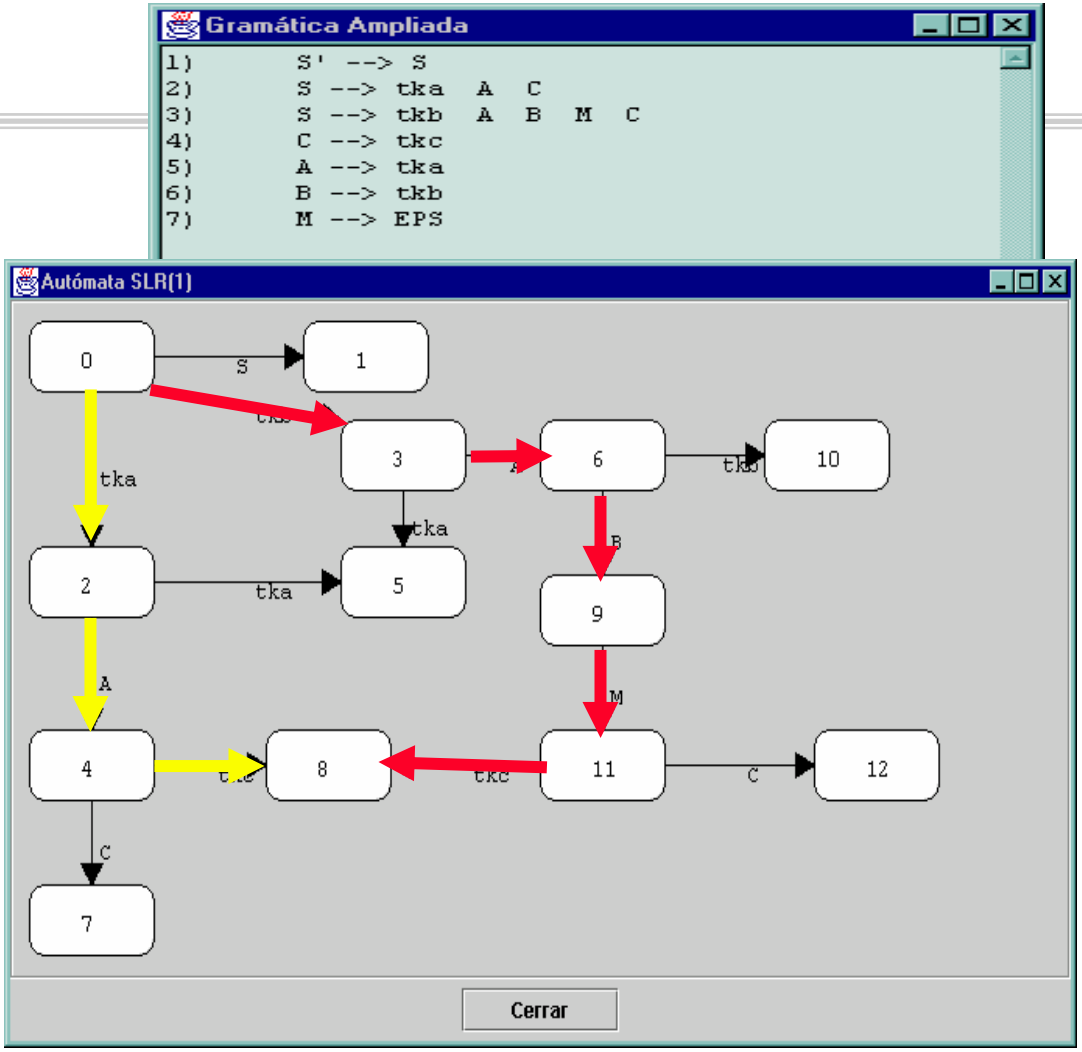
- Una posible solución:
  - introducir marcadores, de manera que la información se encuentre siempre, para todo uso de un mismo símbolo, en las mismas posiciones de la pila

```
S: tka A C      {C.h=A.s}
S: tkb A B M C  {M.h=A.s;C.h=M.s}
C: tkc          {C.s=f(C.h)}

M: ε            {M.s=M.h}
A: tka          {A.s=...}
B: tkb          {B.s=...}
```



Evaluación ascendente de atributos heredados



Evaluación ascendente de atributos heredados

- Cuando la herencia es de dcha. a izda.:
  - transformar la gramática de manera que heredados pasen a ser sintetizados

necesario para la tabla de simb.

```
decVar:  listaID ':' tipo;
tipo:    REAL | INTEGER;
listaID: ID
                {IT($1,??);}
       | listaID ID
                {IT($2,??);}
;
```

```
decVar: ID listaID
                {IT($1,$2);}
;
listaID: ID listaID
                {$$=$2;IT($1,$2);}
       | ':' tipo
                {$$=$2;}
;
tipo: REAL      {$$=tpREAL;}
    | INTEGER   {$$=tpINT;}
;
```

- Otra posible solución: utilizar efectos laterales

```
Para cada nombre en L
      IT(nombre,$3)
FPara
```

```
.........
%%
decVar : listaID  ':' tipo   { }
;
tipo :   REAL          {$$=tpREAL;}
     |   INTEGER       {$$=tpINT;}
;
listaID : ID          {anadeDch($1,L);}
        | listaID ID {anadeDch($2,L);}
;
```

Ejemplos

- Vamos a desarrollar una pequeña calculadora para expresiones de reales
- Un ejemplo de uso:

```
constante cx = 1.2 ;
constante cy = 2.1 ;
variables x y ;


x = cx + 27;
y = 64;
x = cy * (x + y);
escribir x;

y = 2.0 * x;
escribir y;
```

```
-> 193.6200
-> 387.2400
```

Ejemplos

```
%token tkCONSTANTE
%token tkVARIABLES
%token tkESCRIBIR
%token tkREAL tkID
%%
programa:
  parteConstantes
  parteVariables
  parteInstruciones
;
parteConstantes:
| decConstantes
;
decConstantes:
  decConstantes
  decConstante
| decConstante
;
```

```
decConstante:
  tkCONSTANTE
  tkID
  '='
  tkREAL
  ';'
;
parteVariables:
| tkVARIABLES
  decVariables
  ';'
;
decVariables:
  decVariables tkID
| tkID
;
parteInstruciones:
  parteInstruciones
  instruccion
| instruccion
;
```

```
instruccion:
  instAsignacion ';'
| instEscritura  ';'
;
instEscritura:
  tkESCRIBIR
  expr
;
instAsignacion:
  tkID
  '='
  expr
;
```

Ejemplos

```
expr:
  term
| '+' term
| '-' term
| expr opSum term
;
opSum:
  '+'
| '-'
;
term:
  factor
| term opMul factor
;
```

```
opMul:
  '*'
| '/'
;
factor:
  tkID
| tkREAL
| '(' expr ')'
;
```

Ejemplos

```
separadores  ([\t ])+
digito       [0-9]
letra        [a-zA-Z]
constEntera  {digito}+
constReal    {constEntera} ("."{constEntera})?
identificador {letra}({digito}|{letra})*
%%
{separadores}   {linepos+=yyleng;}
"variables"     {trata(tkVARIABLES);}
"constante"     {trata(tkCONSTANTE);}
"escribir"      {trata(tkESCRIBIR);}
{identificador} {strcpy(yylval.laCad,
                        yytext);
                 trata(tkID);}
{constReal}     {sscanf(yytext,"%f",
                         &yylval.elVal);
                 trata(tkREAL);}
\n              {linepos = 0;lineno++;}
.               {trata(yytext[0]);}
%%
```

```
#include "y.tab.h"
#define trata(t) linepos+=yyleng; \
         return((t));

....

void yyerror(char *m)....
%}
```

```
#ifndef TABLA_H
#define TABLA_H

#define MAX_SIMB 100
#define MAX_LONG_ID 30
```

la TS:implement.

```
typedef char identificador[MAX_LONG_ID];
typedef enum {CONSTANTE,VARIABLE} claseDeSimbolo;

typedef struct SIMBOLO{
  identificador elID;
  int estaInicializado;   /*¿valor?*/
  claseDeSimbolo laClase;
  float elValor;
} simbolo, *ptSimbolo;
typedef struct TABLA{
  int numSimbolos;
  simbolo losSimbolos[MAX_SIMB];
} tabla, *ptTabla;
```

Ejemplos

la TS: manejo de un simb.

```
/*--------observacion-----------------*/
#define EL_ID(s) ((s).elID)
#define ESTA_INICIALIZADO(s)   \
                ((s).estaInicializado)
#define CLASE_SIMBOLO(s) ((s).laClase)
#define EL_VALOR(s) ((s).elValor)
/*--------construccion-----------------*/
#define PON_EL_ID(s,id) \
       (strcpy((s).elID,(char *) id))
#define PON_ESTA_INICIALIZADO(s,v) \
       ((s).estaInicializado = (int)(v))
#define PON_CLASE_SIMBOLO(s,c) \
       ((s).laClase = (claseDeSimbolo) c)
#define PON_EL_VALOR(s,v) \
       ((s).elValor = v)
```

la TS: operadores

```
void creaTabla(ptTabla);
   /*----------------------------------------
    Pre:
    Post: t es una tabla vacia
   ----------------------------------------*/
ptSimbolo insertaSimbolo(ptTabla,simbolo);
   /*----------------------------------------
    Pre:
    Post: Si cabe, lo inserta, devolviendo un
          puntero al valor insertado en la
          tabla. Si no cabe, devuelve NULL
   ----------------------------------------*/
ptSimbolo buscaSimbolo(ptTabla,identificador);
   /*----------------------------------------
    Pre:
    Post: Si "s IN T", dev un puntero a S;
          Si_No, dev NULL
----------------------------------------*/
#endif
```

Ejemplos

```
%token tkCONSTANTE tkVARIABLES
%token tkESCRIBIR
%token <elVal>tkREAL
%token <laCad>tkID
%type <elVal>expr
%type <elVal>term
%type <elVal>factor
%type <elVal>term
%type <elOp>opMul
%type <elOp>opSum
%type <elVal>instAsignacion
%%
```

```
#include .....
tabla laTabla;
char mensError[100];

void errorSemantico(char *m
.....}
%}
%union{
   float elVal;
   char laCad[100];
   char elOp;
}
```

Ejemplos

```
programa:
 {creaTabla(&laTabla);}
  parteConstantes
  parteVariables
  parteInstruciones;
parteConstantes:
|     decConstantes;
decConstantes:
  decConstantes
  decConstante
| decConstante;
```

```
decConstante:
  tkCONSTANTE
  tkID
  '='
  tkREAL
  ';'
    { simbolo s,*pS;
      pS = buscaSimbolo(&laTabla,$2);
      if(pS != NULL){
        sprintf(mensError,
          "Simbolo ya existente: %1s",$2);
        errorSemantico(mensError);
      }else{
        PON_EL_ID(s,$2);
        PON_ESTA_INICIALIZADO(s,1);
        PON_CLASE_SIMBOLO(s,CONSTANTE);
        PON_EL_VALOR(s,$4);
        pS = insertaSimbolo(&laTabla,s);
        if(pS == NULL){
          ......
```

Ejemplos

```
parteVariables:
| tkVARIABLES
  decVariables ';'
;
decVariables:
  decVariables
  tkID

|   ......
```

```
{ simbolo s,*pS;
  pS = buscaSimbolo(&laTabla,$2);
  if(pS != NULL){
    sprintf(mensError,
       "Simbolo ya existente: %1s",$2);
    errorSemantico(mensError);
  }else{
    PON_EL_ID(s,$2);
    PON_ESTA_INICIALIZADO(s,0);
    PON_CLASE_SIMBOLO(s,VARIABLE);
    pS = insertaSimbolo(&laTabla,s);
    if(pS == NULL){
      sprintf(mensError,
          "Problemas al insertar simbolo:
          %1s",$2);
      errorSemantico(mensError);
    }
  }
}
```

```
| tkID
   { simbolo s,*pS;
     pS = buscaSimbolo(&laTabla,$1);
     if(pS != NULL){
       sprintf(mensError,"Simbolo ya existente: %1s",$1);
       errorSemantico(mensError);
     }else{
       PON_EL_ID(s,$1);
       PON_ESTA_INICIALIZADO(s,0);
       PON_CLASE_SIMBOLO(s,VARIABLE);
       pS = insertaSimbolo(&laTabla,s);
       if(pS == NULL){
         sprintf(mensError,
            "Problemas al insertar simbolo: %1s",$1);
         errorSemantico(mensError);
       }
     }
   }
;
......
```

Feb. 96

Se desea implementar una calculadora con sintaxis Lisp. El programa debe procesar una secuencia de listas Lisp tomadas de stdin. Cada lista contiene una expresión aritmética (en notación prefija), y para cada una de ellas se debe escribir el resultado de su expresión correspondiente. Para simplificar, asumiremos '+' y '*' como únicos operadores.

```
(+ 2 4 6 8 10);
(* (+ 1 2 3) (+ 9 7 1)) ;
(+   (* 10 2 3)
     25
     (* 8 9 (+ 1 1) )
 );
(+ 20 -20);
(+ 20);
(* 30);
```



```
-> 30
-> 102
-> 229
-> 0
-> 20
-> 30
```

Ejercicio 1 (1 pto.): Escribir un fuente Lex (Flex) para el reconocimiento de los tokens fundamentales que permitan resolver, en conjunción con los Ejercicios 2 y 3, la calculadora deseada.

Ejercicio 2 (2 ptos.): Escribir una gramática (en Yacc/Bison) que exprese una sintaxis (clara y concisa) para las entradas a la calculadora propuesta.

Ejercicio 3 (2 ptos.): Completar la gramática del ejercicio anterior de manera que implemente la calculadora. Notar que basta con añadir al resultado del Ejercicio 2 las acciones necesarias.

Nota 1: Si se considera necesario, se puede establecer un nivel de anidamiento máximo en las listas de listas de 125 niveles

30-I-98

Ejercicio 1 (3.0 ptos.) Considerar :

```
%token tTPENTERO tTPCARACTER tTPBOOLEANO
%token tIDENTIFICADOR tVAL tREF
%union{.....}

parametrosFormales:
|  '(' listaParametros ')'
;
listaParametros:
   listaParametros ';' parametros
|  parametros
;
parametros: claseParametros declaracionParametros
;
declaracionParametros:
   listaID ':' tipo
;
```

30-I-98

```
listaID:
   listaID ‘,’ tIDENTIFICADOR
|  tIDENTIFICADOR
;
tipo:
   tTPENTERO
|  tTPCARACTER
|  tTPBOOLEANO
;
claseParametros: tVAL | tREF
;
```

Ejercicio

Asumimos definidos los siguientes tipos:

```
typedef enum {VALOR,REFERENCIA} clasesDeParametros;
typedef enum {INT,BOOL,CHAR} tiposDeVariables;
typedef .... simbolo,*ptSimbolo;
```

Se dispone también del siguiente procedimiento para insertar un parámetro en la tabla de símbolos:

```
ptSimbolo *insertaParametro(
                  TABLA *laTabla,
                   char *elID,
                   clasesDeParametros laClase,
                   tiposDeVariables elTipo);
```

Se pide definir los atributos y escribir las acciones que se consideren oportunas para insertar los parámetros en la tabla.

Ejercicio

```
typedef enum {VALOR,REFERENCIA} clasesDeParametros;
typedef enum {INT,BOOL,CHAR} tiposDeVariables;

%union{
   struct{
      char elNombre[30];
      clasesDeParametros laClase;
      tiposDeVariables elTipo;
   }paraID;
   clasesDeParametros laClase;
   tiposDeVariables elTipo;
}
%type <paraID> tkID
%type <elTipo> tipo
%type <laClase> claseParametros
%type <elTipo> restoDec
```

Ejercicio: Considerar la siguiente gramática (denominada G), que forma parte de una gramática para un lenguaje de programación imperativo:

```
%token ID OPAS VAL
%%
sentencias:
      sentencia
|     sentencias sentencia
;
sentencia:
      etiqueta asignacion
;
etiqueta:

|     ID ':'
;
asignacion:
      ID OPAS VAL ';'
;
%%
```

04-VII-97

Ejercicio

1) Determinar si $\varepsilon \in L(G)$. Si la respuesta es afirmativa, encontrar una derivación para ella. Si es negativa, razonar la respuesta.
2) Obtener completa la tabla del análisis LL(1), y deducir a partir de ella que no se trata de una gramática de dicha clase
3) Realizar las transformaciones necesarias de la gramática G hasta encontrar una gramática equivalente que sí sea LL(1).
4) Para la gramática obtenida en el punto 3), escribir en C un analizador descendente recursivo. Asumir la existencia de una función

```
int yylex()
```

que realiza el análisis léxico: devuelve los tokens declarados, salta separadores (blancos, tabuladores y saltos de línea) y devuelve el ASCII de los caracteres que no corresponden a los casos anteriores.

- **Ejercicio 2 (3 ptos.):** Con el objetivo de depurar programas se ha decidido implementar un formateador para ficheros que contienen listas tipo Lisp de enteros sin signo. Se pide escribir un analizador sintáctico, en YACC, que recorra el fichero de entrada (contiene una secuencia de listas de las que estamos considerando) y las escriba en forma tabulada por niveles, de acuerdo con el ejemplo. Así, para una línea con la siguiente entrada,

  (1 2 3 4 (5 6) 7 8)

  la salida por *stdout* del analizador debe ser la siguiente

```
(
    1
    2
    3
    4
    (
        5
        6
    )
    7
    8
)
```

Ejercicio

- **Nota**: definir únicamente el token **`'tkNUMERO'`** y asumir ya construído un analizador léxico que:
  - devuelve dicho token cuando reconoce un entero sin signo
  - los blancos, tabuladores y saltos de línea los salta
  - para cualquier otro carácter, devuelve su ASCII