Análisis Sintáctico (III)

Diseño de Lenguajes de Programación

Ingeniería Informática

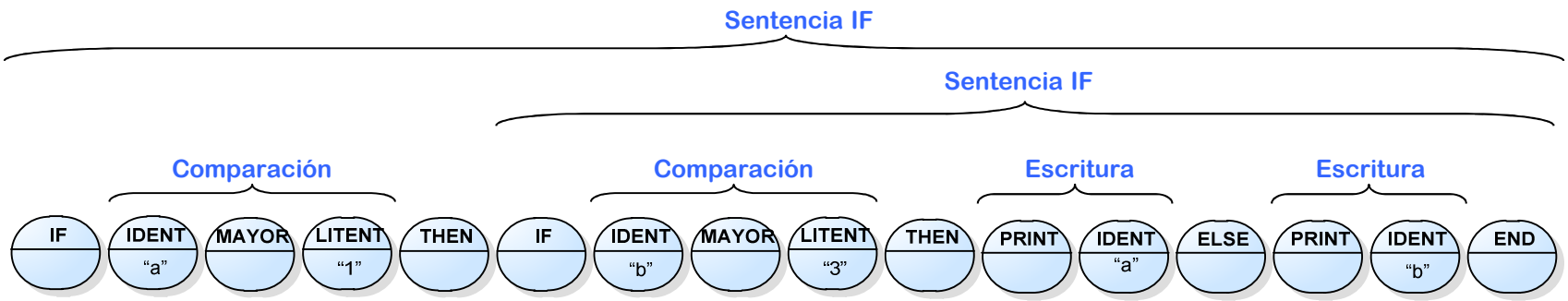
Universidad de Oviedo

(v1.10)

Raúl Izquierdo Castanedo

Funciones

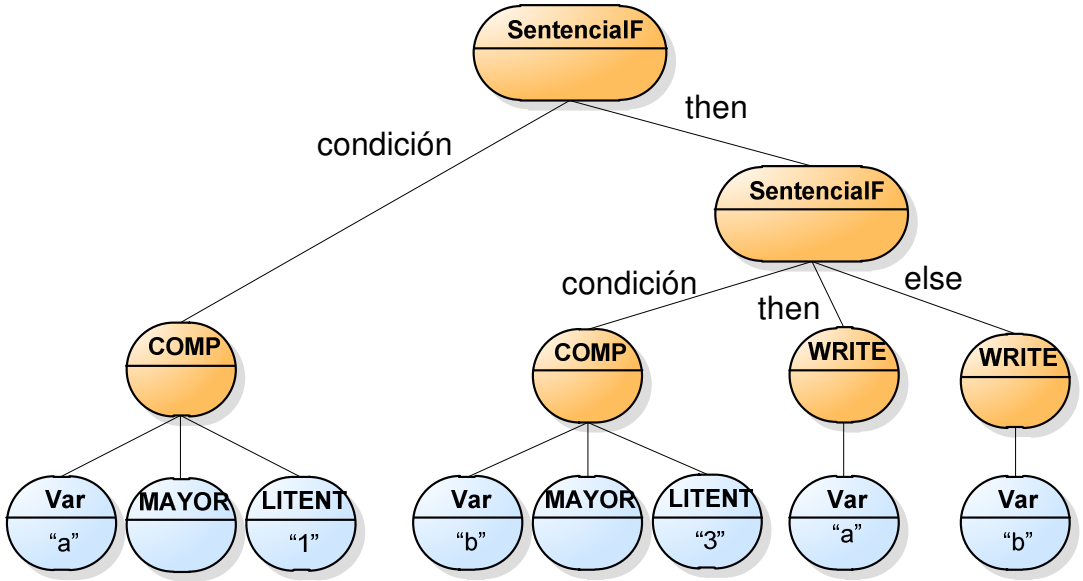
1. Reconocer estructuras (Analizador)



2. Crear árbol

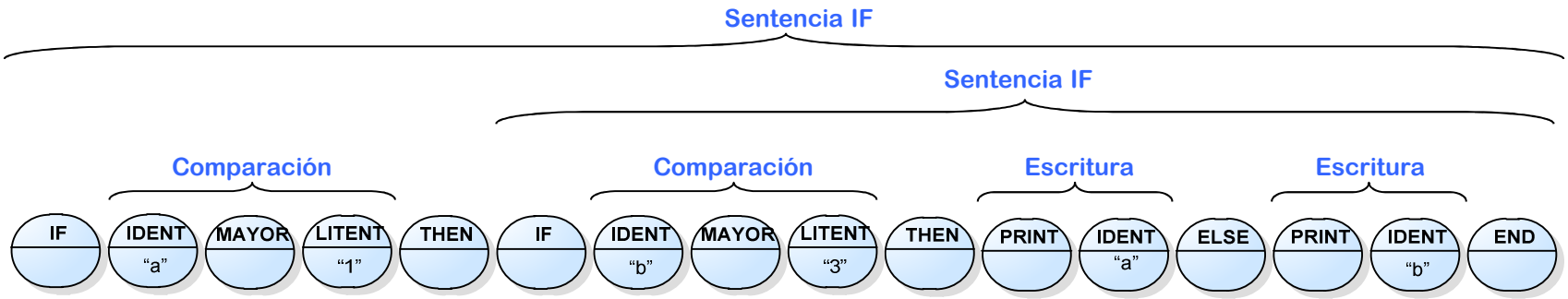
Usted está aquí

El acceso por posición no es práctico

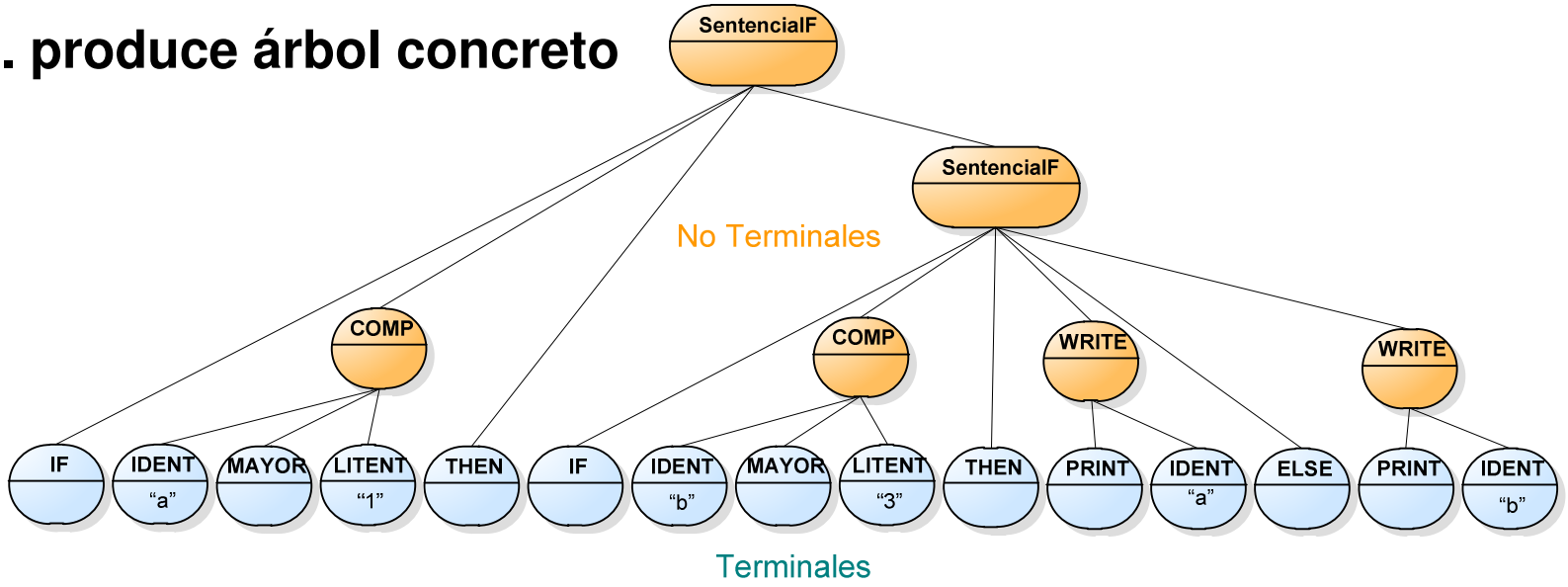


¿Cómo crear el árbol?

Registrar transformaciones...



... produce árbol concreto



Problemas de los árboles de análisis sintácticos (concretos)

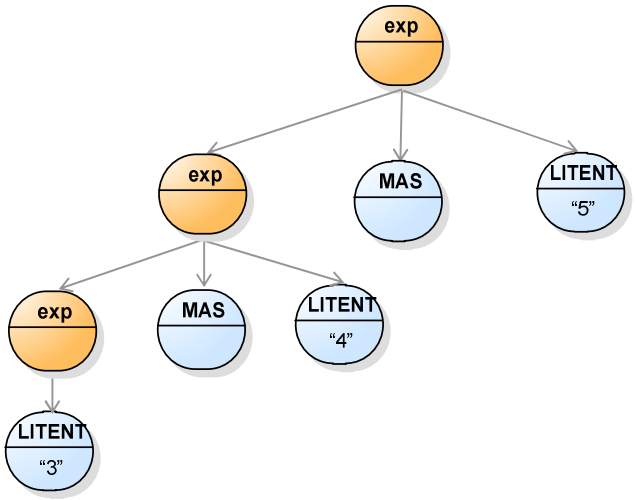
Problema 1

Problemas de los árboles sintácticos (concretos)

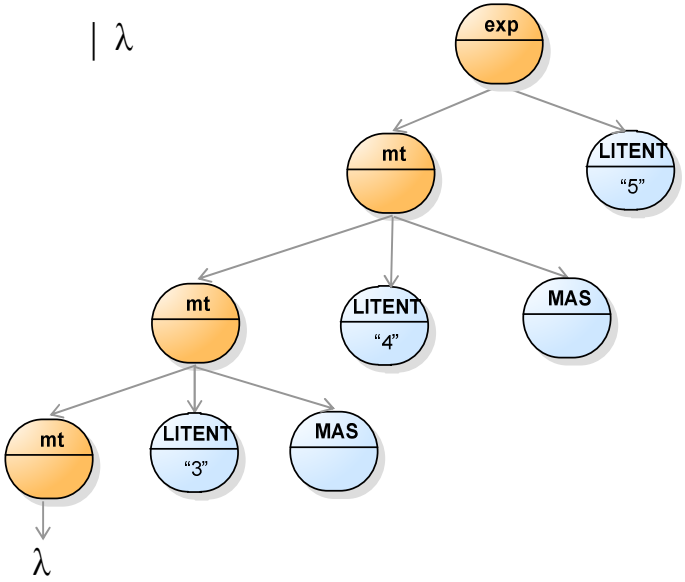
- Dependencia de la GLC

3 + 4 + 5

exp → exp + LITENT
| LITENT



exp → mt LITENT
mt → mt LITENT +
| λ



- Consecuencia: acoplamiento
  - ¿De qué debería depender el árbol?

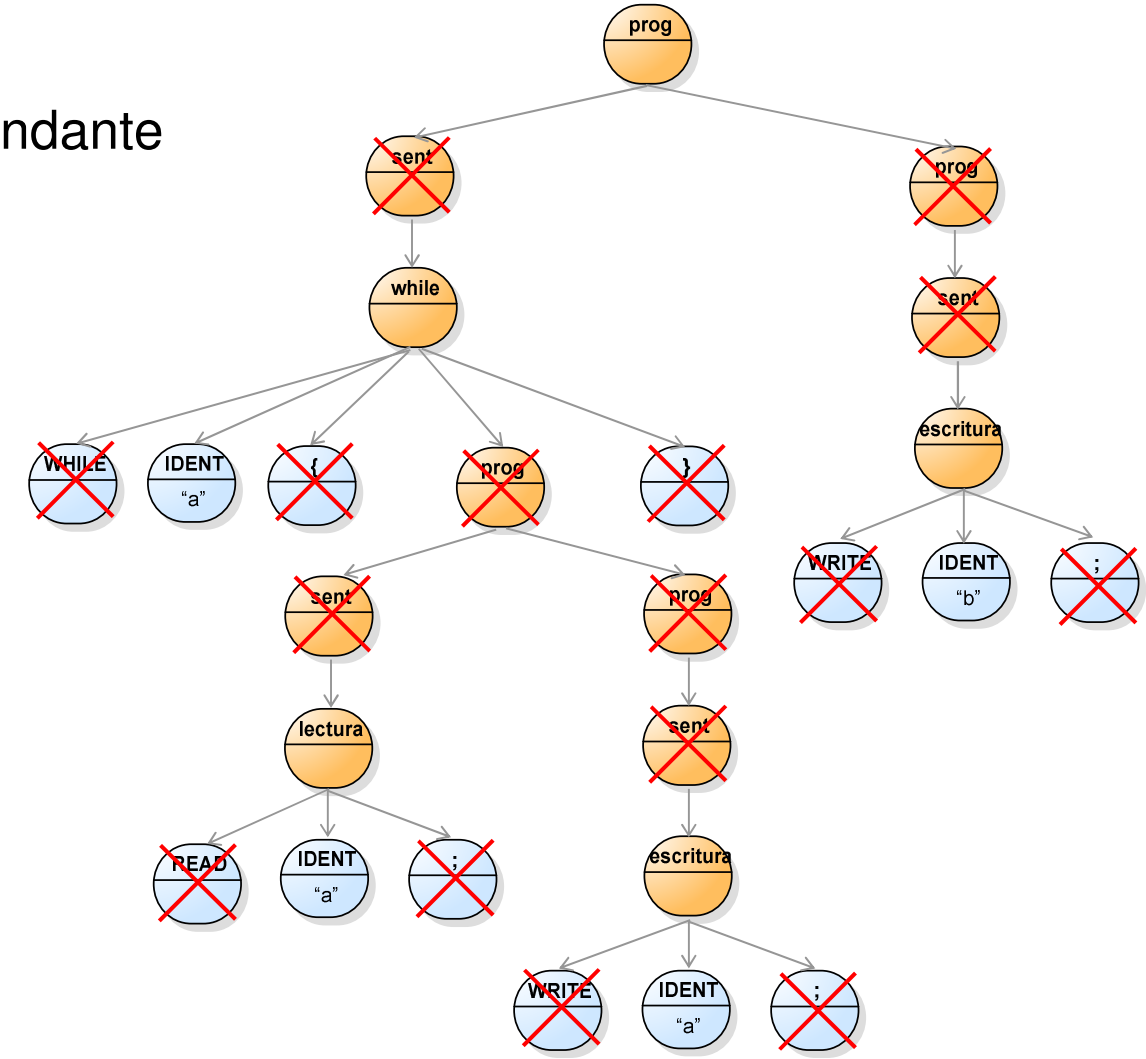
Problema 2

Problemas

- Información redundante

prog → sent | sent prog
sent → lectura | escritura | while
lectura → READ ID ‘;’
escritura → WRITE ID ‘;’
while → WHILE ID ‘{‘ prog ‘}’

```
while a {
   read a;
   write a;
}
write b;
```



Árbol de Sintáxis Abstracta
AST (Abstract Syntax Tree)

AST (I)

Árbol AST

- Estructura independiente de la gramática
- Árbol mínimo que preserva la semántica de la entrada

Catálogo de Nodos

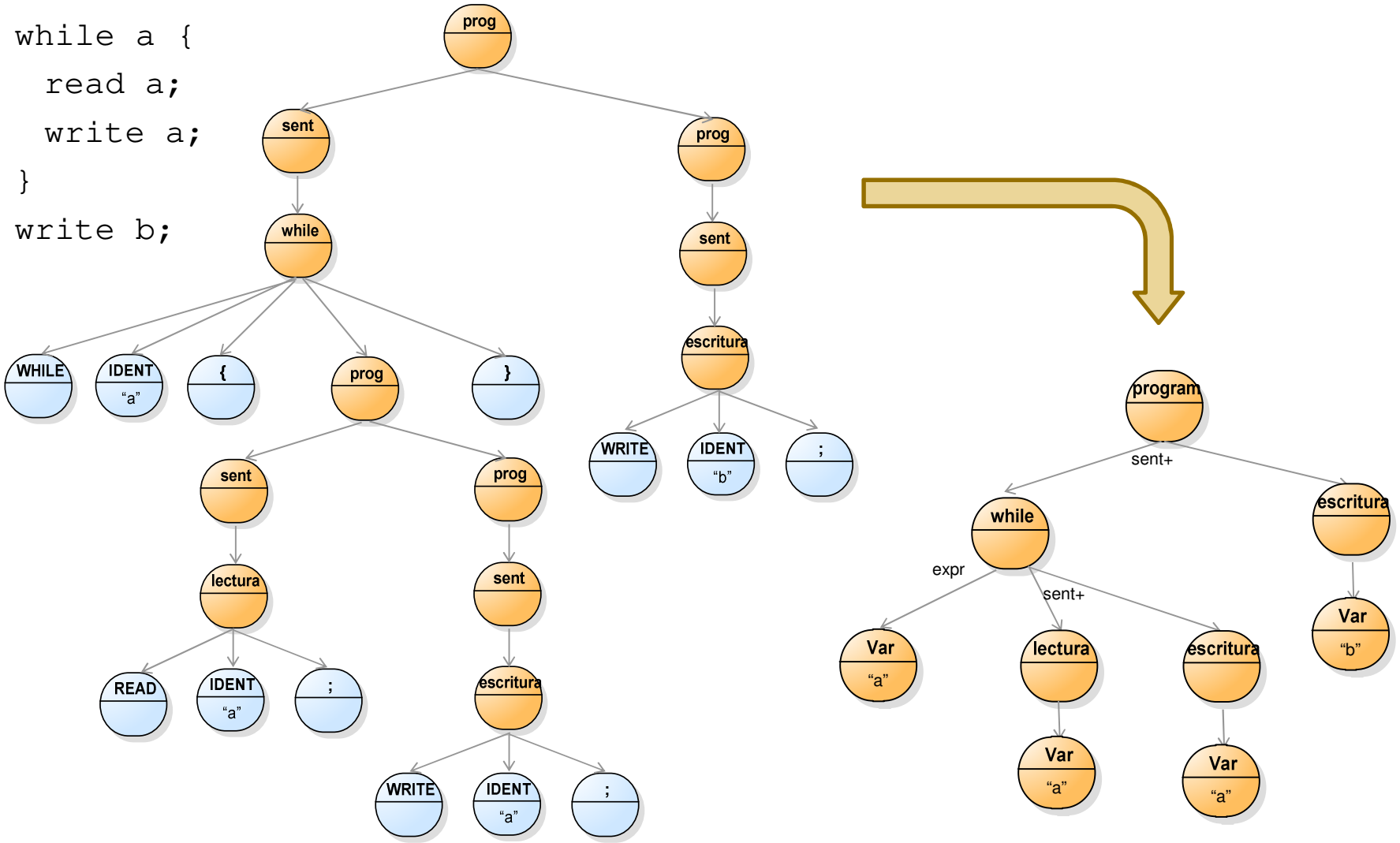
- Conjunto de nodos con los que se construirá un árbol
  - Deben representar a cualquier programa de entrada
- Es un ejercicio de Diseño Orientado a Objetos
  - Los nodos son clases relacionadas mediante Composición
  - El dibujo de un AST es un diagrama de objetos de UML

Criterio de Diseño

- ¿Se puede obtener a partir de la GLC?
- Aquel diseño que facilite el trabajo de las siguientes fases
  - Se ~~podrá cambiar~~ cambiará al llegar a éstas

En ejemplo anterior…

```
while a {
  read a;
  write a;
}
write b;
```



Ejemplo (I)

Obtención del Diseño de los nodos del AST

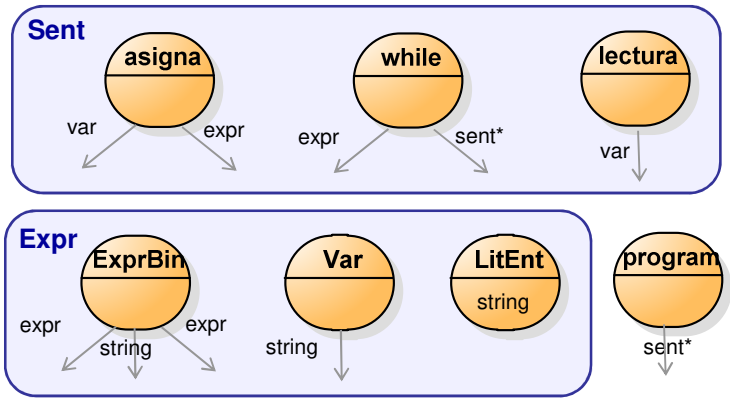
- No asignación múltiple
- Read de una variable
- Condiciones de tipo entero

```
a = 2;
while (a < 100) {
 read b;
 a = a + b ;
}
```

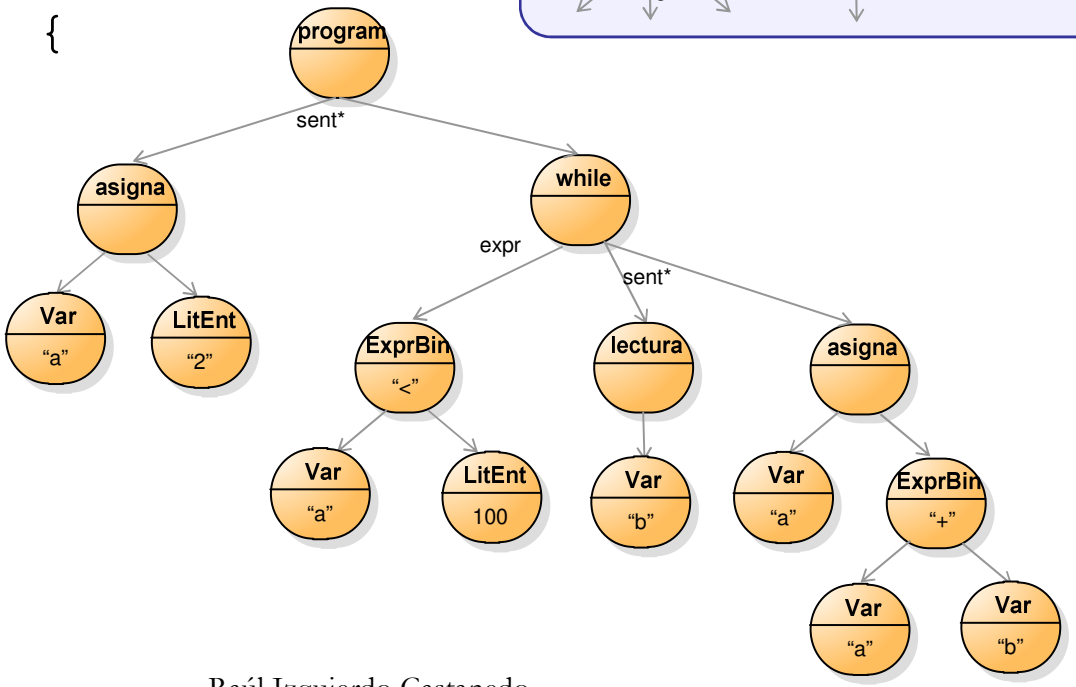
Ejemplo (II)

Obtención del Diseño de los nodos del AST

- No asignación múltiple
- Read de una variable
- Condiciones de tipo entero




```
a = 2;
while (a < 100) {
  read b;
  a = a + b ;
}
```



Metalenguaje. Gramática Abstracta

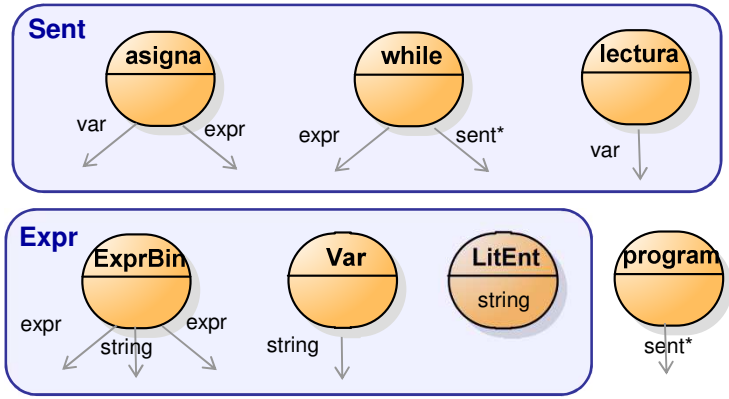
Metalenguaje. Gramática Abstracta (GAb)

Gramática Abstracta (GAb)

- Notación *textual* para describir los nodos con los que se forman los árboles
  - Lo necesitarán las demás fases

nodo[:categorias] → [nombre$_i$:]tipo$_i$[*]

Ejemplo



programa → sentencia*

asigna:sentencia → variable expr
while:sentencia → expr sentencia*
lectura:sentencia → variable

exprBinaria:expr → left:expr operador:string right:expr
variable:expr → nombre:string
literalEntero:expr → valor:string

Generación de las clases de un AST

Traducción a código

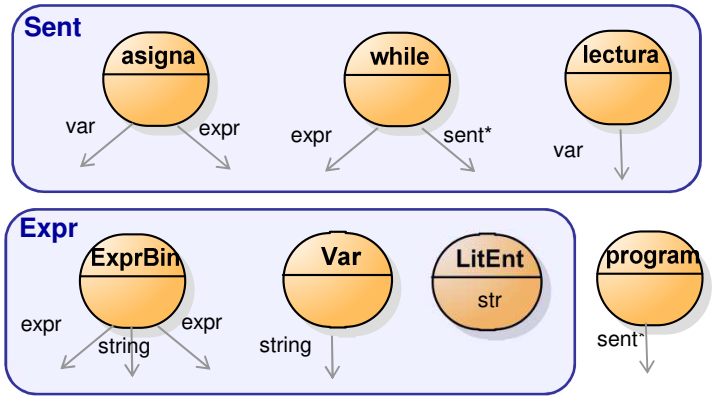
- Interfaz AST
- Interfaz por cada categoría
- Una Clase por Nodo
  - Implementa interfaces de las categorías a las que pertenezca
  - Un atributo por cada rama/descendiente
    - Si la rama tiene * se define una List

Implementación de un AST

programa → sentencia*

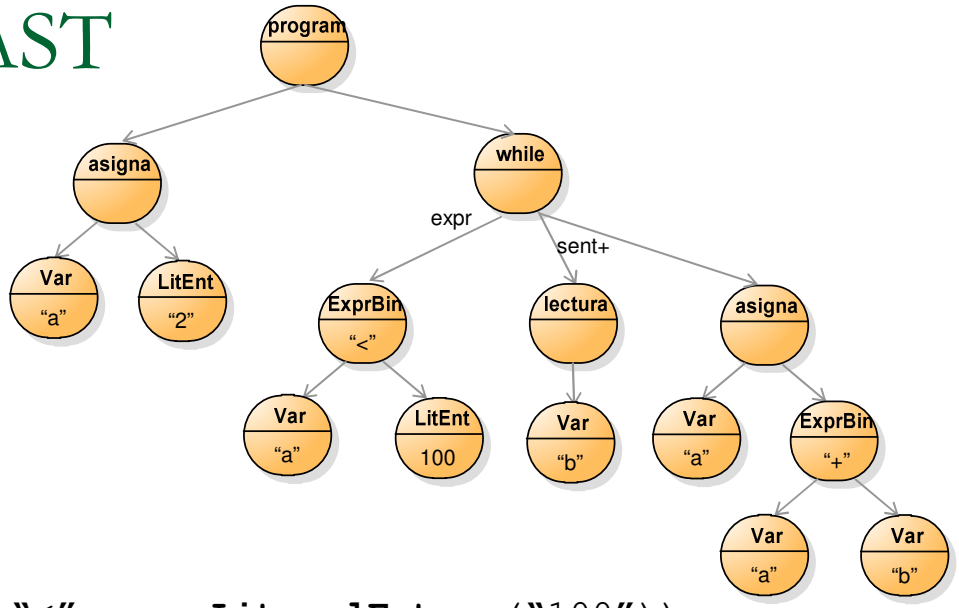
asigna:sentencia → variable expr
while:sentencia → expr sentencia*
lectura:sentencia → variable

exprBinaria:expr → left:expr
  operador:string right:expr
variable:expr → nombre:string
literalEntero:expr → valor:string




```
interface AST { }

interface Sentencia extends AST  { }
interface Expr extends AST  { }

class Programa implements AST {
  List<Sentencia> sentencias;
}

class Asigna implements Sentencia {
  Variable var;
  Expr expr;
}
class While implements Sentencia {
  Expr expr;
  List<Sentencia> sentencias;
}
class Lectura implements Sentencia {
  Variable var;
}

class ExprBinaria implements Expr {
  Expr left, right;
  String operador;
}
class Variable implements Expr {
  String nombre;
}
class LiteralEntero implements Expr {
  String valor;
}
```

Implementación de un AST




```
Programa prog = new Programa();

Asigna asigna1 = new Asigna(
  new Variable("a"),
  new LiteralEntero("2"));
prog.sentencias.Add(asigna1);

While while = new While();
while.cond =
  new ExprBinaria(new Variable("a"), "<", new LiteralEntero("100"));

Lectura read = new Lectura(new Variable("b"));
while.sentencias.Add(read);

ExprBinaria suma = new ExprBinaria(new Variable("a"), "+", new
    Variable("b"));
Asigna asigna2 = new Asigna(new Variable("a"), suma);
while.sentencias.Add(asigna2);

prog.sentencias.Add(while);
```

Ejercicio E1

Dado el siguiente lenguaje

- Tipos entero y double
- Condiciones de tipo entero
- Las definiciones antes de las sentencias
- Puede haber varias sentencias en cada rama del if

Obtener

1. Diseñar los nodos del AST y expresarlos mediante una Gramática Abstracta
2. Implementar la Gramática Abstracta en Java
3. Dibujar el AST de la entrada siguiente

```
a:double;

if (a > 2) then
  write a;
else
  write g(5, a) + 8.3;
endif

write f(a * 2);
```

Tarea

Diseñar nodos AST

```
type Persona: record {
  edad:int;
  meses:[12] int;
}
ap: [20] Persona;

doble(a:int, b:int): real {
  local:int;
  local = a + b;
  return local * 2.0;
}
edad(p:Persona):real {
  return (real) p.edad;
}
escribeNumero(num:int) {
  write ap[num].edad;
}
```

```
/* continúa aquí */
main() {
  local: int;
  pepe: Persona;

  pepe.edad = 20;
  pepe.sueldo = 1;

  local = 0;
  while (local < 10) {
    a[local] = doble(2, local);
    local = local + 1;
  }

  write edad(pepe);
  escribe(pepe.meses[2]);
}
```

Generar Parsers con Yacc

Análisis Sintáctico

Acciones en Yacc

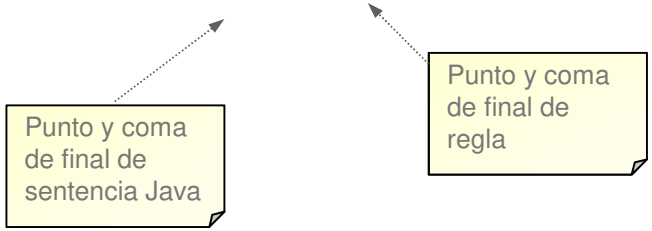
Código generado por YACC

- Al acabar el análisis (*yyparse*) y volver al *main* se han perdido todos los símbolos obtenidos de la entrada
  - Se han ido descartando de la pila a medida que se hacían reducciones

Yacc permite intervenir durante el proceso de reconocimiento

- Tercer tipo de símbolo
  - Acciones: código en Java
- Se ejecutan en las reducciones

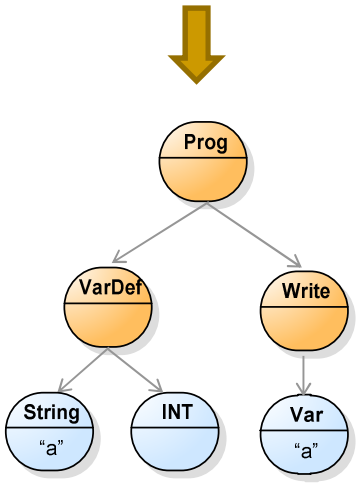
```
a: b IDENT c { sop("Encontrado programa"); } ;
```



Aplicación de las Acciones

Lo que se necesita…

```
int a;
print a;
```



... y dichos nodos deberían crearse en la regla adecuada

```
programa: definicion sentencia { … = new Programa(…); }
    ;

definicion: tipo IDENT     { … = new VarDef(…); } ;

tipo: INT { … = new IntType(…); }
  | REAL  { … = new RealType(…); }
  ;

sentencia: PRINT expr      { … = new Print(…); }

expr: LITENT       { … = new LiteralEntero(…); }
    | IDENT        { … = new Variable(…); }
    ;
```

Pero...

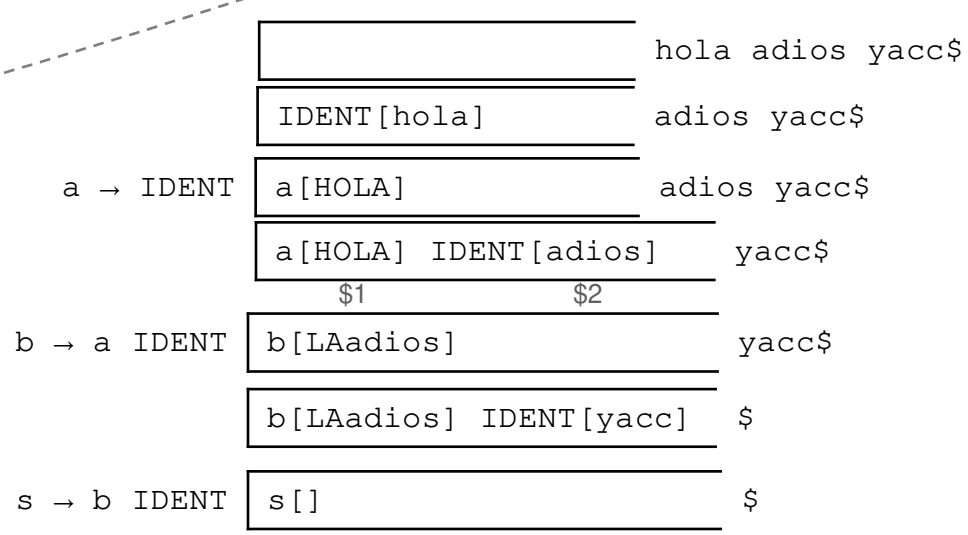
- ¿Dónde se dejan los objetos creados?
- ¿De dónde se toman los parámetros del constructor?

Atributos en Yacc

Al introducir un símbolo en la pila se puede guardar un valor con él

- Para guardar un valor con un...
  - ... token: se deja en la variable *yylval* (lo hace yylex)
  - ... no-terminal: se deja en la variable *$$*
- Para extraer el valor metido con un símbolo
  - usar la variable *$n* donde n es la posición del símbolo en la regla

```
// Entrada: hola adios yacc
%%
s: b IDENT          { sop($1 + ":" + $2); } ;

b: a IDENT          { $$ = ((String)$1).substring(2) + $2; } ;

a: IDENT            { $$ = ((String)$1).toUpperCase(); } ;
%%

int yylex() {
  try {
    int token = lex.yylex();
    yylval = lex.lexeme();
    return token;
  } catch (Exception e) {
    return -1;
  }
}
…
```

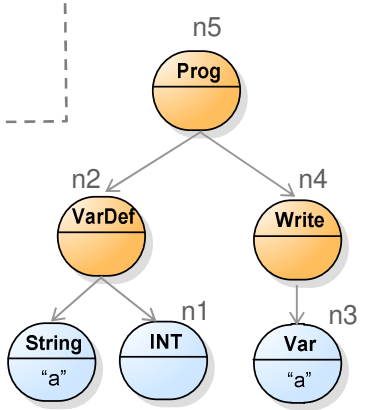


Ejemplo

Ejemplo anterior completo

```
%%
programa: definicion sentencia { raiz = new Programa($1, $2); } ;

definicion: tipo IDENT { $$ = new VarDef($1, $2); } ;

tipo: INT { $$ = new IntType(); }
  | REAL  { $$ = new RealType(); }
  ;
sentencia: PRINT expr   { $$ = new Print($2); }

expr: IDENT { $$ = new Variable($1); }
      | LITENT{ $$ = new LiteralEntero($1); }
      ;
%%
AST raiz;
…
```

n5, n2, n4, n1, n3



Salida

Entrada:

```
int a
print a
```

		int a print a $
	INT[int]	a print a $
tipo → INT	tipo[n1]	a print a $
	tipo[n1] IDENT[a]	print a $
def → tipo IDENT	def[n2]	print a $
	def[n2] PRINT	a $
	def[n2] PRINT IDENT[a]	$
expr → IDENT	def[n2] PRINT expr[n3]	$
sent → PRINT expr	def[n2] sent[n4]	$
prog → def sent	prog[]	$

Ejercicio E2

Crear Parser del lenguaje que genera la siguiente gramática

```
%token GATO PERRO
%%
s: oracion;

oracion: sujeto verbo objeto '.';

sujeto: TODO nombre
   | EL nombre
   | PEPE
  ;

nombre: GATO | PERRO;

verbo:  MUERDE | TREPA A

objeto: LA CORTINA
   | UN ARBOL
   |
  ;
%%
```

```
interface AST {}

class Oracion implements AST {
  Sujeto sujeto;
  Verbo verbo;
  Objeto objeto;
}

interface Sujeto extends AST { }
class Todo implements Sujeto {
  Nombre nombre;
  Todo (Object nombre) {
    this.nombre = (Nombre) nombre;
  }
}
class El implements Sujeto {
  Nombre nombre;
  El (Object nombre) {
    this.nombre = (Nombre) nombre;
  }
}
class Pepe implements Sujeto { }

interface Nombre extends AST { }
class Perro implements Nombre { }
class Gato implements Nombre { }

interface Verbo extends AST { }
class Muerde implements Verbo { }
class Trepa implements Verbo { }

interface Objeto extends AST { }
class Cortina implements Objeto { }
class Arbol implements Objeto { }
class Nada implements Objeto { }
```

Tarea E3

Lenguaje cuyas sentencias están formadas por

- Una lista
- Cada lista puede tener cero o mas elementos
- Cada elemento puede ser un número o una lista

Ejemplos de entrada

```
()
(34)
(34, 22, 45)
(12, (), 66, (34), (23,45,67))
(12, ( ( (12, 23), 54, 67), 15, 22), 5)
```

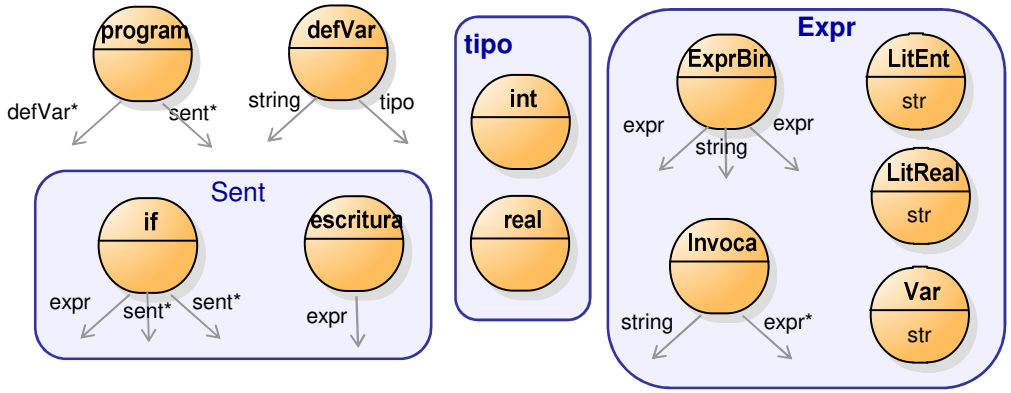
- Generar Parser que devuelva AST que represente la estructura de la lista leída

Soluciones

Análisis Sintáctico

Solución E1 (I)

```
a:double;

if (a>2) then
  write a;
else
  write g(5, a) + 8.3;
endif
write f(a * 2);
```



Gramática Abstracta

programa → defVariable* sentencia*
defVariable → nombre:string tipo

intType:tipo → λ
realType:tipo → λ

escritura:sentencia → expresion
if:sentencia → condicion:expresion cierto:sentencia* falso:sentencia*

exprBinaria:expresion → left:expresion operator:string right:expresion
invocacion:expresion → nombre:string argumentos:expresion*
variable:expresion → lexema:string
literalInt:expresion → lexema:string
literalReal:expresion → lexema:string

Solución E1 (II)

Gramática Abstracta

programa → defVariable* sentencia*
defVariable → nombre:string tipo

intType:tipo → λ
realType:tipo → λ

escritura:sentencia → expresion
if:sentencia → condicion:expresion cierto:sentencia* falso:sentencia*

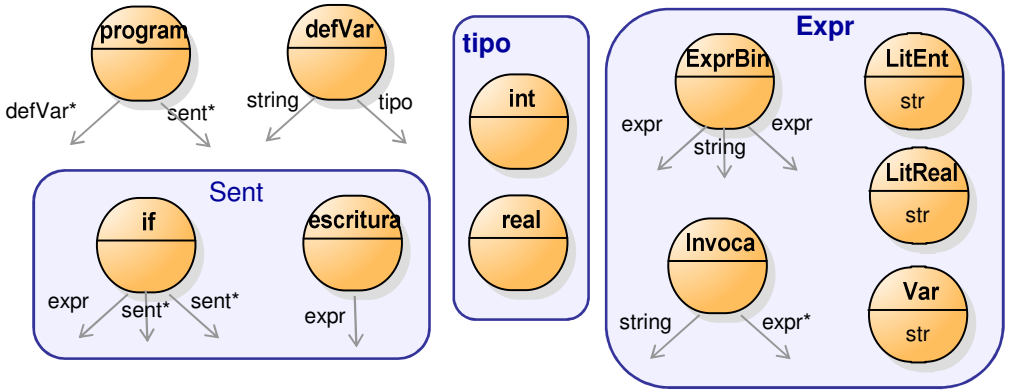
exprBinaria:expresion → left:expresion operator:string right:expresion
invocacion:expresion → nombre:string argumentos:expresion*
variable:expresion → lexema:string
literalInt:expresion → lexema:string
literalReal:expresion → lexema:string

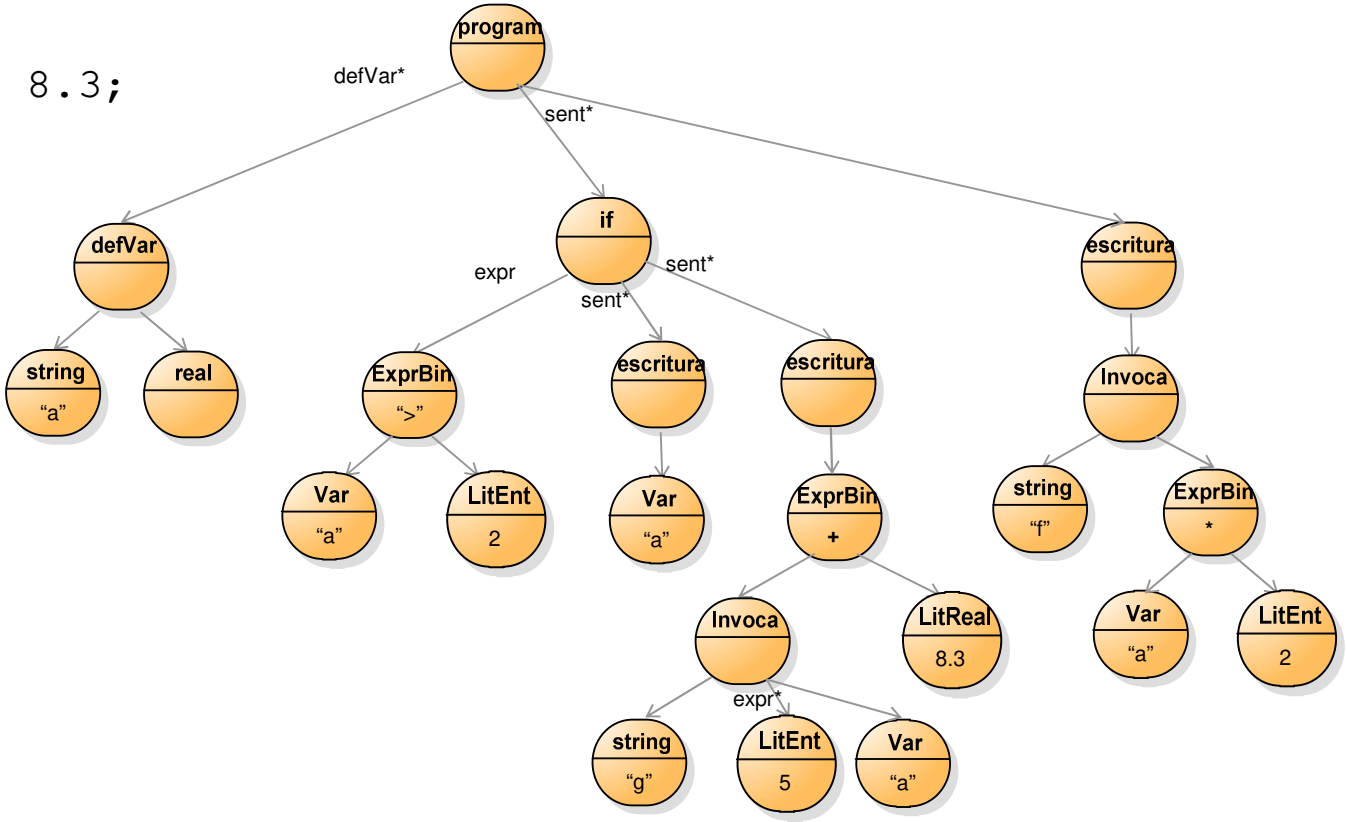
Implementación en Java

```
interface AST { }

class Programa implements AST {
    List<DefVariable> variables;
    List<Sentencia> sentencias;
}
class DefVariable implements AST {
    String nombre;
    Tipo tipo;
}

interface Tipo extends AST { }
class IntType implements Tipo { }
class RealType implements Tipo { }

interface Sentencia extends AST {}

class Excritura implements Sentencia {
    Expresion expresion;
}
class If implements Sentencia {
    Expr condicion;
    List<Sentencia> cierto;
    List<Sentencia> falso;
}

interface Expresion extends AST { }

class ExprBinaria implements Expresion {
    Expresion left, right;
    String operator;
}
class Invoca implements Expresion {
    String nombre;
    List<Expresion> argumentos;
}
class Variable implements Expresion {
    String lexema;
}
class LiteralEntero implements Expresion {
    String lexema;
}
class LiteralReal implements Expresion {
    String lexema;
}
```

Solución E1 (III)

```
a:double;

if (a>2) then
  write a;
else
  write g(5, a) + 8.3;
endif
write f(a * 2);
```





Solución E2

```
%%
s: oracion { oracion = (Oracion)$1; } ;
oracion: sujeto verbo objeto '.' { $$ = new Oracion($1, $2, $3); } ;

sujeto: TODO nombre        { $$ = new Todo($2); }
  | EL nombre              { $$ = new El($2); }
  | PEPE          { $$ = new Pepe(); }
  ;
nombre: GATO      { $$ = new Gato(); }
    | PERRO       { $$ = new Perro(); }
    ;
verbo: MUERDE     { $$ = new Muerde(); }
    | TREPA A     { $$ = new Trepa(); }
   ;
objeto: LA CORTINA         { $$ = new Cortina(); }
  | UN ARBOL               { $$ = new Arbol(); }
  |                        { $$ = new Nada(); }
  ;
%%
Oracion oracion;
Oracion getAST() { return oracion; }
```

Solución E3

```
%%
s: serie { arbol = (AST) $1; } ;

serie: '(' listaOpt ')' { $$ = $2; } ;

listaOpt: lista
  |             { $$ = new Serie(); }
  ;

lista: e        { $$ = new Serie($1); }
  | lista ',' e { $$ = $1; ((Serie)$$).add($3); }
  ;

e: LITENT { $$ = new LiteralEntero($1); }
  | serie
  ;
%%
```