



Tema 7

Análisis semántico en compiladores



- 7.1 La semántica de los lenguajes de programación de propósito general
- 7.2 Expresiones
- 7.3 Instrucciones
- 7.4 Funciones, estructuras de datos y objetos
- 7.5 El sistema de tipos





- El diseño de analizadores semánticos es una herramienta muy útil dentro del desarrollo de software:
 - Para procesar cualquier fichero de datos almacenado en un formato descrito por un lenguaje formal se necesita un analizador semántico.
 - Ejemplos de formatos descritos por lenguajes formales:
 - HTML (hipertexto)
 - XML (información etiquetada)
 - ARFF (conjuntos de datos)
 - XPM (gráficos)
 - TEX (documentos)
 - ...
 - Para almacenar los diseños de una herramienta CAD
 - Para importar datos de versiones anteriores o de herramientas externas
 - Cualquier transacción de datos descrita por medio de un lenguaje formal



- El diseño de compiladores es el origen de los analizadores semánticos.
- En un compilador, la entrada a analizar es un fichero escrito en un lenguaje de programación.
- El objetivo del analizador semántico, dentro de la estructura del compilador, es generar el árbol de sintaxis abstracta con toda la información del fichero analizado.
- Las etapas posteriores del compilador convierten esta estructura de datos en una descripción en código ensamblador.



- Los lenguajes de programación se pueden clasificar de varias formas:
 - Imperativos / declarativos
 - De propósito general / de propósito específico
 - De alto nivel / de bajo nivel
 - Estructurados / no estructurados
 - Orientados a objetos / orientados a proceso
 - Fuertemente tipados / débilmente tipados
- Aspectos comunes en los lenguajes de programación imperativos de propósito general:
 - Expresiones
 - Sentencias
 - Estructuras de datos, funciones, objetos.
 - Sistema de tipos



- Características:
 - Definen un proceso que permite calcular un valor mediante una sucesión de operaciones básicas.
 - Se representan mediante una estructura de árbol.
 - El tipo de datos que genera es conocido.
- Ejemplo de expresiones:
 - Literales (125, 0x4FFF, 2578356L, 3.4, 0.5e-10, 'A', "HOLA MUNDO")
 - Variables (x, var, aux125)
 - Operaciones unarias (!x, -x)
 - Operaciones binarias (x+y, x&y, x&&y, x>>3)
 - Operaciones ternarias (inicio? 0 : 1)
 - Llamadas a funciones (pow(2.0, 3.5))
 - Referencia a un elemento de un array (x[2], y[0][3][7])
 - Referencia al campo de una estructura (alumno.nombre)

• Estructura de datos en C

- Estructura única con todos los campos posibles y un código para expresar el tipo de expresión

```
typedef struct {  
    int code;  
    int integerLiteral;  
    float floatLiteral;  
    Expr* left;  
    Expr* right;  
    ...  
} Expr;
```

- Unión de varias estructuras, cada una de ellas con los campos necesarios y un código para expresar el tipo de expresión.

```
typedef struct {  
    int code;  
    int intLiteral;  
} IntegerLiteral;
```

```
typedef struct {  
    int code;  
    float floatLiteral;  
} FloatLiteral;
```

```
typedef struct {  
    int code;  
    Expr* left;  
    Expr* right;  
} BinaryExpr;
```

```
typedef union {  
    IntegerLiteral intLiteral;  
    FloatLiteral floatLiteral;  
    BinaryExpr binary;  
} Expr;
```

• Estructura de datos en Java

- Clase abstracta que describe una expresión genérica

```
public abstract class Expr {  
    public abstract Type getType();  
}
```

- Clases distintas para los distintos tipos de expresiones

```
public class IntegerLiteral extends Expr {  
    private int value;  
    public IntegerLiteral(int v) { ... }  
    public Type getType() { return Type.integer; }  
}
```

```
public class Binary extends Expr {  
    private int code;  
    private Expr left;  
    private Expr right;  
    public Binary(int code, Expr l, Expr r) { ... }  
    public Type getType() { ... }  
}
```



- Gramáticas:
 - $\text{Expr} \rightarrow \text{LogicExpr} \{ \text{Expr.s} = \text{LogicExpr.s} \}$
 - $\text{LogicExpr} \rightarrow \text{AndExpr}_1 \{ \text{LogicExpr.s} = \text{AndExpr}_1.s \}$
 $(\text{OR } \text{AndExpr}_2 \{ \text{LogicExpr.s} = \text{new Binary}(\text{OR}, \text{LogicExpr.s}, \text{AndExpr}_2.s); \})^*$
 - $\text{AndExpr} \rightarrow \text{NotExpr}_1 \{ \text{AndExpr.s} = \text{NotExpr}_1.s \}$
 $(\text{AND } \text{NotExpr}_2 \{ \text{AndExpr.s} = \text{new Binary}(\text{AND}, \text{AndExpr.s}, \text{NotExpr}_2.s); \})^*$
 - $\text{NotExpr} \rightarrow \text{NOT } \text{CompExpr} \{ \text{NotExpr.s} = \text{new Unary}(\text{NOT}, \text{CompExpr.s}); \}$
 $\mid \text{CompExpr} \{ \text{NotExpr.s} = \text{CompExpr.s}; \}$
 - $\text{CompExpr} \rightarrow \text{AritExpr}_1 \{ \text{CompExpr.s} = \text{AritExpr}_1.s; \}$
 $(\text{CompOp } \text{AritExpr}_2$
 $\{ \text{CompExpr.s} = \text{new Binary}(\text{Comp.s}, \text{CompExpr.s}, \text{AritExpr}_2.s); \})^?$
 - $\text{CompOp} \rightarrow \text{EQ} \{ \text{CompOp.s} = \text{EQ} \}$
 $\mid \text{GT} \{ \text{CompOp.s} = \text{GT}; \}$
 $\mid \dots$



- Gramáticas:
 - $\text{AritExpr} \rightarrow (\text{MINUS} \{ \text{minus} = \text{true}; \})^? \text{TermExpr}_1$
 $\{ \text{AritExpr.s} = \text{minus? } \text{new Unary}(\text{MINUS}, \text{ermExpr}_1.s) : \text{TermExpr}_1.s; \}$
 $(\text{AddOp } \text{TermExpr}_2$
 $\{ \text{AritExpr.s} = \text{new Binary}(\text{AddOp.s}, \text{AritExpr.s}, \text{TermExpr}_2.s); \})^*$
 - $\text{AddOp} \rightarrow \text{PLUS} \{ \text{AddOp.s} = \text{PLUS} \} \mid \text{MINUS} \{ \text{AddOp.s} = \text{MINUS} \}$
 - $\text{TermExpr} \rightarrow \text{FactorExpr}_1 \{ \text{TermExpr.s} = \text{FactorExpr}_1.s \}$
 $(\text{ProductOp } \text{FactorExpr}_2$
 $\{ \text{TermExpr.s} = \text{new Binary}(\text{ProductOp.s}, \text{TermExpr.s}, \text{FactorExpr}_2.s); \})^*$
 - $\text{ProductOp} \rightarrow \text{PROD} \{ \text{ProductOp.s} = \text{PROD}; \}$
 $\mid \text{DIV} \{ \text{ProductOp.s} = \text{DIV}; \}$
 $\mid \text{MOD} \{ \text{ProductOp.s} = \text{MOD}; \}$



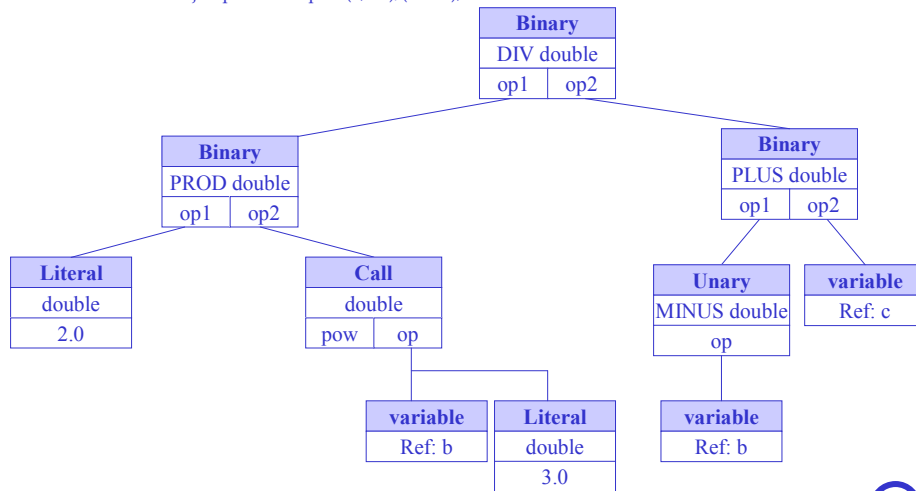
- Gramáticas:
 - FactorExpr \rightarrow LPAREN Expr RPAREN { FactorExpr.s = Expr.s; }
 - | Literal { FactorExpr.s = Literal.s; }
 - | ID { FactorExpr.s = new VariableExpr(ID.lex); }
 - | ID LBRACKET Expr RBRACKET
{ Factor.Expr.s = new Element(ID.lex,Expr.s); }
 - | ID LPAREN { Vector<Expr> v = new Vector<Expr>(); }
(Expr₁ {v.add(Expr₁.s);} (COMMA Expr₂ {v.add(Expr₂.s);})*)?
RPAREN { FactorExpr.s = new Call(ID.lex,v); }



- Gramáticas:
 - Literal \rightarrow INT_LITERAL { Literal.s = new IntLiteral(INT_LITERAL.lex); }
 - | LONG_LITERAL { Literal.s = new LongLiteral(LONG_LITERAL.lex); }
 - | FLOAT_LITERAL { Literal.s = new FloatLiteral(FLOAT_LITERAL.lex); }
 - | DOUBLE_LITERAL { Literal.s = new DoubleLiteral(DOUBLE_LITERAL.lex); }
 - | CHAR_LITERAL { Literal.s = new CharLiteral(CHAR_LITERAL.lex); }
 - | STRING_LITERAL { Literal.s = new StringLiteral(STRING_LITERAL.lex); }
 - | TRUE { Literal.s = new BooleanLiteral(true); }
 - | FALSE { Literal.s = new BooleanLiteral(false); }
 - | NULL { Literal.s = new NullLiteral(); }

• Gramáticas:

- Ejemplo: $2.0 * \text{pow}(x, 3.0) / (-b + c)$;



• Verificaciones semánticas:

- Verificaciones de tipos (p.e. no se pueden sumar expresiones booleanas)
- Verificaciones de variables y campos (deben estar declaradas previamente)
- Verificaciones de funciones y métodos (deben existir)



- Describen el flujo de control de las operaciones que se llevan a cabo en una función.
- Tipos de instrucciones:
 - Declaraciones
 - Asignaciones
 - Condicionales: if – then – else, switch-case
 - Bucles: while, do-while, for
 - Saltos: break, continue, return, throw
 - Bloques de instrucciones
 - Tratamiento de errores: try-catch-finally



- Estructura de datos en C
 - Estructura única con todos los campos posibles y un código para expresar el tipo de instrucción

```
typedef struct {
    int code;
    Expr cond;
    Statement* next;
    Statement* body;
    Statement* elseStm;
    ...
} Statement;
```

- Unión de varias estructuras, cada una de ellas con los campos necesarios y un código para expresar el tipo de instrucción.

```
typedef struct {
    int code;
    Expr cond;
    Statement* thenStm;
    Statement* elseStm;
} IfStatement;
```

```
typedef struct {
    int code;
    Expr cond;
    Statement* body;
} WhileStatement;
```

```
typedef struct {
    int code;
    Variable* left;
    Expr* right;
} AssignStatement;
```

```
typedef union {
    IfStatement ifStm;
    WhileStatement whileStm;
    AssignStatement assignStm;
} Statement;
```

- Estructura de datos en Java
 - Clase abstracta que describe una instrucción genérica

```
public abstract class Statement {  
    }  
}
```

- Clases distintas para los distintos tipos de instrucciones

```
public class IfStatement extends Statement {  
    private Expr cond;  
    private Statement thenStm;  
    private Statement elseStm;  
  
    public IfStatement(Expr c, Statement t,  
        Statement e) { ... }  
}
```

```
public class WhileStatement extends Statement {  
    private Expr cond;  
    private Statement body;  
  
    public WhileStatement(Expr c, Statement b) {  
        ...  
    }  
}
```

- Instrucciones (gramática)
Instruccion → Decl | Asign | IfThen | SwitchCase | While | DoWhile | For
| Break | Continue | Return | Bloque
- Instrucciones (atributos)
 - Tabla de símbolos (ToS): es un atributo heredado en todas las instrucciones. Almacena las variables declaradas, los argumentos de las funciones, las variables y constantes globales, etc.
 - Instrucción (inst): es un atributo sintetizado que contiene la estructura de datos asociada a la instrucción reconocida



Declaraciones

- Gramática:
$$\text{Decl} \rightarrow \text{Tipo ID} \{ \text{Var } v = \text{Decl.ToS.add}(\text{Tipo.t}, \text{ID.lexema}); \text{Decl.inst} = \text{new Block}(); \}$$
$$(\text{EQ Expr} \{ \text{Decl.inst.add}(\text{new AssignStm}(v, \text{Expr.s});) \})?$$
$$(\text{COMMA ID} \{ v = \text{Decl.ToS.add}(\text{Tipo.t}, \text{ID.lexema}); \}$$
$$(\text{EQ Expr} \{ \text{Decl.inst.add}(\text{new AssignStm}(v, \text{Expr.s});) \})? \}^* \text{PYC}$$
- Verificaciones semánticas
 - Hay que evitar variables duplicadas
 - El tipo de la variable y de la expresión debe ser compatible
- Acciones semánticas
 - La declaración de una variable produce su inserción en la tabla de símbolos
 - Si se admite la declaración con asignación, el atributo *inst* debe contener la instrucción de asignación.
 - Si se admite la declaración de una lista de variables, las asignaciones formarán un bloque de instrucciones.



Asignaciones

- Gramática:
$$\text{Asign} \rightarrow \text{ID} \{ \text{Var } v = \text{Asign.ToS.search}(\text{ID.lexema}); \}$$
$$\text{EQ Expr} \{ \text{Asign.inst.} = \text{new AssignStm}(v, \text{Expr.s}); \} \text{PYC}$$
- Verificaciones semánticas:
 - La variable debe existir
 - El tipo de la variable y de la expresión debe ser compatible

Instrucción IF-THEN-ELSE

- Gramática:
IfThen → IF LPAREN Expr RPAREN Instrucción
{ IfThen.inst = new IfStm(Expr.s, Instrucción.inst); }
(ELSE Instrucción { *IfThen.inst.setElse(Instrucción.inst);* })?
- Verificaciones semánticas
 - La expresión debe ser booleana

Instrucción SWITCH-CASE

- Gramática:
SwitchCase → SWITCH LPAREN Expr RPAREN LBRACE
{ SwitchCase.inst = new SwitchCaseStm(Expr.s); }
(Case { *SwitchCase.inst.addCase(Case.clause);* }
| Default { *SwitchCase.inst.addDefault(Default.clause);* }) *
RBRACE
Case → CASE Valor DOSP { *Case.clause = new CaseClause(Valor.v);* }
(Instruccion { *Case.clause.add(Instrucción.inst);* }) *
Default → DEFAULT DOSP { *Default.clause = new DefaultClause();* }
(Instruccion { *Default.clause.add(Instrucción.inst);* }) *
- Verificaciones semánticas:
 - La expresión debe ser entera
 - Los valores de las cláusulas CASE no se pueden repetir
 - No puede haber más de una cláusula DEFAULT

Instrucción WHILE

- Gramática:
 - While → WHILE LPAREN Expr RPAREN Instrucción
 - { While.inst = new WhileStm(Expr.s, Instrucción.inst); }*
- Verificaciones semánticas
 - La expresión debe ser booleana

Instrucción DO-WHILE

- Gramática:
 - DoWhile → DO Instrucción WHILE LPAREN Expr RPAREN PYC
 - { DoWhile.inst = new DoWhileStm(Expr.s, Instrucción.inst); }*
- Verificaciones semánticas
 - La expresión debe ser booleana

Instrucción FOR

- Gramática:
 - For → FOR LPAREN ForInit PYC Expr PYC ForUpdate RPAREN Instrucción
 - { For.inst = new ForStm(ForInit.inst, Expr.s, ForUpdate.inst, Instrucción.inst); }*
 - ForInit → { ForInit.inst = new Bloque(); }
 - (Tipo ID { Variable v = ForInit.ToS.add(Tipo.t, ID.lexema); }
 - (EQ Expr { ForInit.inst.add(v, Expr.s) }) ?
 - (COMA ID { v = ForInit.ToS.add(Tipo.t, ID.lexema); }
 - (EQ Expr { ForInit.inst.add(v, Expr.s) }) ?) *) ?
 - ForUpdate → { ForUpdate.inst = new Bloque(); }
 - (ID EQ Expr { Variable v = ForUpdate.ToS.search(ID.lexema); }
 - ForUpdate.inst.add(new AssignStm(v, Expr.s)); }
 - (COMMA ID EQ Expr { v = ForUpdate.ToS.search(ID.lexema); }
 - ForUpdate.inst.add(new AssignStm(v, Expr.s)); }) *) ?
- Verificaciones semánticas: la condición debe ser una expresión booleana

Instrucciones de salto incondicional

- Gramática:
 - Break → BREAK PYC { *Break.inst = new BreakStm();* }
 - Continue → CONTINUE PYC { *Continue.inst = new ContinueStm();* }
 - Return → RETURN (Expr)? PYC { *Return.inst = new ReturnStm(Expr.s);* }
 - Throw → THROW Expr PYC { *Throw.inst = new ThrowStm(Expr.s);* }
- Verificaciones semánticas
 - La instrucción break debe encontrarse dentro de un bucle o de un switch
 - La instrucción continue debe encontrarse dentro de un bucle
 - El tipo de datos de la expresión de un return debe corresponder al tipo de la función en la que se encuentre
 - El tipo de dato de la expresión de un throw debe ser una referencia a una excepción o a un error.

Bloques de instrucciones

- Gramática:
 - Bloque → { *Bloque.inst = new Bloque();* }
 - LBRACE (Instrucción { *Bloque.inst.add(Instrucción.inst);* })^{*} RBRACE
- Verificaciones semánticas
 - No se puede añadir ninguna instrucción detrás de un salto incondicional (break, continue, return, throw).

Instrucción TRY-CATCH

- Gramática:

```
TryCatch → TRY Bloque { TryCatch.inst = new TryStm(Bloque.inst); }  
          ( CatchClause { TryCatch.inst.addClause(CatchClause.clause); } ) *  
          ( FinallyClause { TryCatch.inst.addClause(FinallyClause.clause); } ) ?  
CatchClause → CATCH LPAREN Tipo ID RPAREN Bloque  
             { CatchClause.ToS.add(Tipo.t, ID.lexema);  
               CatchClause.clause = new CatchClause(Bloque.inst); }  
FinallyClause → FINALLY Bloque { FinallyClause.clause = new FinallyClause(Bloque.inst); }
```

- Verificaciones semánticas

- El tipo debe ser una referencia a un error o excepción
- No puede haber dos cláusulas catch con el mismo tipo
- Debe haber al menos una cláusula catch o finally

- Componentes de los lenguajes orientados a procesos

- Variables y constantes globales
- Definición de estructuras de datos (typedef)
- Funciones

- Variables y constantes globales

- Se pueden tratar como declaraciones.
- Se incluyen en la Tabla de Símbolos en el ámbito más general

- Definiciones de estructuras de datos

- Se incluyen en la Tabla de Símbolos.
- Las definiciones de tipos se describen en el próximo apartado

- Funciones

- Se incluyen en la Tabla de Símbolos
- Se identifican por su nombre y por los tipos de datos de sus argumentos
(En C sólo se utiliza el nombre)



- Componentes de los lenguajes orientados a objetos
 - Clases e interfaces
 - Su definición se incluye en la Tabla de Símbolos
- Características de los lenguajes orientados a objetos
 - Encapsulado
 - Herencia
 - Polimorfismo
- Componentes de una clase
 - Campos, métodos y constructores
 - En Java, también puede ser código estático y nuevas clases o interfaces
 - Clase de la que heredan (o clases, si se soporta herencia múltiple)
 - Interfaces que implementan



- Estructura para representar un campo
 - Modificadores de acceso
 - Tipo de datos
 - Identificador
 - Expresión para calcular su valor por defecto
- Estructura para representar un método
 - Modificadores de acceso
 - Tipo de datos que devuelve
 - Identificador
 - Argumentos
 - Lista de instrucciones



- Estructura para representar un constructor
 - Modificadores de acceso
 - Identificador
 - Argumentos
 - Lista de instrucciones
- Verificaciones semánticas
 - Los campos no pueden estar repetidos
 - Los constructores deben tener el mismo nombre que la clase
 - Los argumentos de un método o constructor no pueden repetirse
 - Los constructores y métodos no pueden tener el mismo nombre y tipo de argumentos (duplicados) pero sí pueden tener el mismo nombre (polimorfismo)
 - Las clases con miembros abstractos deben ser abstractas.



- Se trata de uno de los aspectos a desarrollar en el análisis semántico
- Objetivos:
 - Verificar que el tipo de una construcción coincide con el previsto en su contexto
 - Resolver la sobrecarga de operadores
 - Resolver el polimorfismo
 - Determinar las transformaciones de tipo
- La comprobación de tipos se basa en el “sistema de tipos” del lenguaje, que debe describirse en detalle en su definición



- Expresiones de tipos
 - Tipos básicos de lenguaje (int, long, float, double, char, ...)
 - Tipo básico especial (error_tipo)
 - Matrices:
 - Expresión de tipo
 - Rango
 - Productos: producto cartesiano de expresiones de tipo (int x int)
 - Registro: Producto en el que se asocia un nombre a cada expresión
 - Punteros: Apuntador a una expresión de tipo
 - Funciones:
 - Dominio: Producto de expresiones de tipo
 - Rango: Expresión de tipo



- Sistemas de tipos
 - Conjunto de reglas para asignar expresiones de tipo a las diferentes partes de un programa
- Comprobador de tipos
 - Se suele especificar mediante un analizador dirigido por la sintaxis
- Comprobación estática
 - Parte de la comprobación que se puede realizar en tiempo de compilación
- Comprobación dinámica
 - Parte de la comprobación que debe realizarse en tiempo de ejecución (por ejemplo, los límites de una matriz)



- Comprobación de tipos en expresiones
 - Expr → **true** { Expr.tipo = boolean }
 - Expr → **false** { Expr.tipo = boolean }
 - Expr → **entero** { Expr.tipo = int }
 - Expr → **largo** { Expr.tipo = long }
 - Expr → **real** { Expr.tipo = float }
 - Expr → **realdoble** { Expr.tipo = double }
 - Expr → **caracter** { Expr.tipo = char }
 - Expr → **cadena** { Expr.tipo = array(char) }
 - Expr → **id** { Expr.tipo = Expr.ToS.search(id.lexema).tipo }



- Comprobación de tipos en expresiones
 - Expr → Expr1 **prod** Expr2
 - { if(Expr1.tipo == int && Expr2.tipo == int) Expr.tipo = int;
 - else if((Expr1.tipo == int || Expr1.tipo == long) && (Expr2.tipo == int || Expr2.tipo == long)) Expr.tipo = long;
 - else if((Expr1.tipo == int || Expr1.tipo == long || Expr1.tipo == float) && (Expr2.tipo == int || Expr2.tipo == long || Expr2.tipo == float)) Expr.tipo = float;
 - else if((Expr1.tipo == int || Expr1.tipo == long || Expr1.tipo == float || Expr1.tipo == double) && (Expr2.tipo == int || Expr2.tipo == long || Expr2.tipo == float || Expr2.tipo == double)) Expr.tipo = float;
 - else Expr.tipo = error_tipo; }



- Comprobación de tipos en expresiones

- $Comp \rightarrow igual \mid distinto \mid mayor \mid menor \mid mayor_igual \mid menor_igual$

- $Expr \rightarrow Expr1 \text{ Comp } Expr2$

```
{ if( (Expr1.tipo == int || Expr1.tipo == long ||  
      Expr1.tipo == float || Expr1.tipo == double) &&  
      (Expr2.tipo == int || Expr2.tipo == long ||  
      Expr2.tipo == float || Expr2.tipo == double) ) Expr.tipo = boolean;  
  else Expr.tipo = error_tipo; }
```

- $Expr \rightarrow Expr1 \text{ and } Expr2$

```
{ if( Expr1.tipo == boolean && Expr2.tipo == boolean ) Expr.tipo = boolean;  
  else Expr.tipo = error_tipo; }
```



- Comprobación de tipos en expresiones

- $Expr \rightarrow Expr1 \text{ '[' } Expr2 \text{ '}'$

```
{ if( (Expr2.tipo == int && Expr1.tipo == array(T) ) Expr.tipo = T;  
  else Expr.tipo = error_tipo; }
```

- $Expr \rightarrow \text{apuntador } Expr1 \text{ } \{ \text{if}(Expr1.tipo == puntero(T)) Expr.tipo = T;$

```
  else Expr.tipo = error_tipo; }
```

- $Expr \rightarrow \text{direccion } Expr1 \text{ } \{ Expr.tipo = puntero(Expr1.tipo); \}$

- $Expr \rightarrow Expr1 \text{ '(' } Expr2 \text{ ')'}$

```
{ if( (Expr2.tipo == S && Expr1.tipo == (S→T) ) Expr.tipo = T;  
  else Expr.tipo = error_tipo; }
```

- Comprobación de tipos en instrucciones

- $Stm \rightarrow id \text{ asig } Expr \text{ pyc}$

```
{ if( compatible(buscaTipo(id.lexema),Expr.tipo) ) Stm.tipo = void;  
  else Stm.tipo = error_tipo; }
```

- $Stm \rightarrow \text{if } Expr \text{ then } Stm1$

```
{ if(Expr.tipo == boolean && Stm1.tipo == void ) Stm.tipo = void;  
  else Stm.tipo = error_tipo; }
```

- $Stm \rightarrow \text{while } Expr \text{ do } Stm1$ { if(Expr.tipo == boolean && Stm1.tipo == void)

```
  Stm.tipo = void;  
  else Stm.tipo = error_tipo; }
```

- $Stm \rightarrow Stm1 ; Stm2$

```
{ if(Stm1.tipo == void && Stm2.tipo == void ) Stm.tipo = void;  
  else Stm.tipo = error_tipo; }
```

- Equivalencia de expresiones de tipos

- Las expresiones de tipo se construyen a partir de tipos básicos y constructores
- Equivalencia estructural: dos expresiones son equivalentes si tienen la misma estructura, es decir, si se forman a partir de los mismos tipos básicos aplicando los mismos constructores.
- Equivalencia de nombre: dos expresiones son equivalentes si tienen el mismo nombre (registros)
- La mayoría de lenguajes optan por la equivalencia estructural
- Ejemplo: struct tipo1 { int dato11; int dato12; }
 struct tipo2 { int dato21; int dato22; }