

# Tema 3. Aproximación a la definición y desarrollo de un traductor sencillo

---

3.1 Introducción

3.2 Definición sintáctica de un lenguaje

3.3 Traducción dirigida por la sintaxis

3.4 Análisis sintáctico

3.5 Traducción de los principales constructores de los L.P.

3.6 Análisis semántico

3.7 La Tabla de Símbolos

3.8 Tipos de código intermedio

3.9 Extensión de la traducción de los principales constructores de los L.P.

## 3.2 Definición sintáctica

---

### **Gramáticas Independientes del Contexto**

1. Un conjunto de símbolos terminales (tokens)
2. Un conjunto de símbolos no-terminales.
3. Un conjunto de producciones, donde cada producción consiste en un no-terminal y una secuencia, que puede ser vacía, de terminales y/o no-terminales.
4. Un no-terminal como símbolo inicial.

# Gramáticas Independientes del Contexto

---

- $G = (T, N, P, S)$
- $P = \{A \rightarrow \alpha \mid A \in N \wedge \alpha \in (T \cup N)^*\}$
- Derivación:  
 $\alpha_1 \Rightarrow \alpha_2$  sii  $\alpha_1 = \beta_1 A \beta_2$  y  $\alpha_2 = \beta_1 \alpha \beta_2$  con  $A \in N$  y  $(A \rightarrow \alpha) \in P$
- Lenguaje generado:  
 $x \in T^*$  tal que  $S \xRightarrow{*} x$

# Ejemplo

---

$G = (\{+, -, *, /, \text{identificador}, \text{entero}\}, \{E\}, P, E)$

$P = \{E \rightarrow E + E, E \rightarrow E - E,$

$E \rightarrow E * E, E \rightarrow E / E,$

$E \rightarrow \text{identificador}, E \rightarrow \text{entero}\}$

# Ejemplo (notación abreviada)

---

$E \rightarrow E + E$

|  $E - E$

|  $E * E$

|  $E / E$

| **identificador**

| **entero**

# Árboles de análisis (de derivación)

---

1. La raíz está etiquetada con el símbolo inicial.
2. Cada hoja está etiquetada bien con un terminal, bien con la cadena vacía.
3. Cada nodo interior está etiquetado con un no-terminal.
4. Si un no-terminal  $A$  etiqueta un nodo interior y  $X_1, X_2, \dots, X_n$  son etiquetas de los hijos de ese nodo, entonces  $A \rightarrow X_1 X_2 \dots X_n$  es una producción. Como caso especial, si  $A \rightarrow \xi$ , entonces un nodo etiquetado  $A$  puede tener un solo hijo, etiquetado  $\xi$ .

# Ambigüedad

---

- Se dice que una gramática es ambigua cuando existe una cadena generada por la gramática a la que corresponden dos árboles de análisis distintos.

# Ejemplo de ambigüedad

---

$E \rightarrow E + E$

|  $E - E$

|  $E * E$

|  $E / E$

| **identificador**

| **entero**



# Gramáticas equivalentes

---

- Dos gramáticas son equivalentes si definen el mismo lenguaje
- Gramática equivalente a G, no ambigua:

$$\begin{aligned} E \rightarrow & E + T \\ & | E - T \\ & | T \end{aligned}$$
$$\begin{aligned} T \rightarrow & T * F \\ & | T / F \\ & | F \end{aligned}$$
$$\begin{aligned} F \rightarrow & \text{identificador} \\ & | \text{entero} \end{aligned}$$

## 3.3 Traducción dirigida por la sintaxis

---

- Esquemas de traducción dirigidas por la sintaxis
- Recorrido en profundidad del árbol de análisis
- Atributos

# Esquemas de traducción dirigidos por la sintaxis

---

- Se construye sobre una gramática independiente del contexto
- Atributos asociados a los componentes sintácticos
  - » Terminales: valor viene dado por el análisis léxico.
  - » No-terminales: valor calculado mediante reglas sem.
- Conjunto de reglas semánticas, asociadas a cada producción
  - » especifican el proceso de traducción (dando valores a los atributos)

# Interpretación de un E.T.D.S.

---

- Construcción del árbol de *análisis decorado*
- Si un nodo **n** está etiquetado con el símbolo de la gramática **X**, escribimos **X.a** para denotar el valor del atributo **a** de **X** en ese nodo.
- El valor de **X.a** en **n** se calcula usando la regla semántica correspondiente

# Interpretación de un E.T.D.S. (II)

---

- Para evaluar un árbol de análisis se construye el árbol de análisis decorado con los valores de los atributos vacíos
- Se dibujan las acciones en el árbol como si fueran un símbolo terminal más.
- El momento en que una acción debe ejecutarse viene dado por el lugar que ocupa en la parte derecha de la producción.

# Recorrido en profundidad

---

- Un recorrido en profundidad de un árbol empieza por la raíz y acaba por las hojas. Este proceso se repite para cada subárbol que forma parte del árbol.
- Trabajaremos con recorridos de **izquierda a derecha**.

```
procedure visitar(n: nodo);  
begin  
  for cada hijo  $m$  de  $n$ , de izquierda a derecha  
    if  $m$  es un nodo then visitar( $m$ )  
    elsif  $m$  es una regla semantica then evaluar( $m$ )  
  end loop  
end
```

# Ejemplo de E.T.D.S.

---

- $E \rightarrow E + E \quad \{E.\text{valor} := E_1.\text{valor} + E_2.\text{valor}\}$
- |  $E - E \quad \{E.\text{valor} := E_1.\text{valor} - E_2.\text{valor}\}$
- |  $E * E \quad \{E.\text{valor} := E_1.\text{valor} * E_2.\text{valor}\}$
- |  $E / E \quad \{E.\text{valor} := E_1.\text{valor} / E_2.\text{valor}\}$
- | **entero**  $\{E.\text{valor} := \text{entero.valor}\}$

## Ejemplo de E.T.D.S. (II)

---

$E \rightarrow E + T$	$\{E.\text{valor} := E_1.\text{valor} + T.\text{valor}\}$
$\quad   E - T$	$\{E.\text{valor} := E_1.\text{valor} - T.\text{valor}\}$
$\quad   T$	$\{E.\text{valor} := T.\text{valor}\}$
$T \rightarrow T * F$	$\{T.\text{valor} := T_1.\text{valor} * F.\text{valor}\}$
$\quad   T / F$	$\{T.\text{valor} := T_1.\text{valor} / F.\text{valor}\}$
$\quad   F$	$\{T.\text{valor} := F.\text{valor}\}$
$F \rightarrow \text{entero}$	$\{F.\text{valor} := \text{entero}.\text{valor}\}$



# Atributos

---

- Se dice que un atributo de un símbolo de la gramática es sintetizado si su valor en un nodo del árbol de análisis se calcula a partir de los atributos de los hijos de ese nodo.
- Atributos léxicos
- Otro tipo de atributos: los atributos heredados

## 3.4 Análisis sintáctico

---

- Análisis descendente
- Análisis predictivo
- Construcción de un analizador predictivo
- Transformación de una gramática en predictiva

# Análisis descendente

---

- El árbol de análisis comienza a construirse por la raíz y se termina en las hojas
- Para cada nodo **n** etiquetado con un no-terminal **A**
  - » seleccionar una de las producciones para **A** y construir los hijos de **n**, que serán los símbolos de la parte derecha de la producción escogida

# Ejemplo

---

tipo  $\rightarrow$  tipo\_simple

|  $\uparrow$ id

| array [ tipo\_simple ] of tipo

tipo\_simple  $\rightarrow$  integer

| char

| num punto punto num

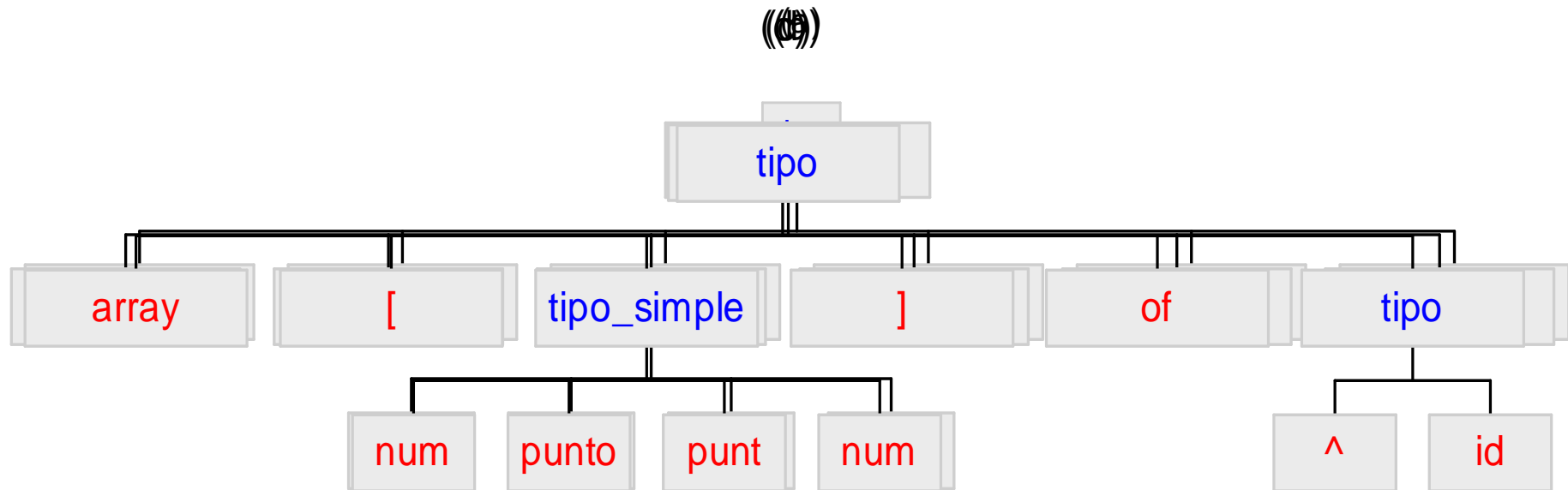
# Análisis predictivo (I)

---

- Para algunas gramáticas es posible implementar los pasos anteriores durante un recorrido de izquierda a derecha de la cadena de entrada
- El token en curso que está siendo examinado en un momento dado se denomina *lookahead*

# array [1..10] of ^ persona

---



# Análisis predictivo (II)

---

- En el análisis predictivo, el lookahead determina sin ambigüedades cuál debe ser la producción elegida para cada no-terminal en un momento dado del proceso de análisis
- **PRIMERO( $\alpha$ )**: conjunto de tokens que aparecen como primeros símbolos de una o más cadenas generadas por  $\alpha$

# Análisis predictivo (III)

---

- $A \rightarrow \alpha$
- $A \rightarrow \beta$
- Si  $\text{PRIMERO}(\alpha)$  y  $\text{PRIMERO}(\beta)$  son disjuntos
  - » el lookahead puede ser utilizado para decidir qué producción utilizar



# Construcción de un analizador predictivo

---

- Un procedimiento para cada no-terminal
  - » Decidir qué producción usar
  - » Ejecutar una serie de acciones construidas miméticamente con respecto a las partes derechas de las producciones
    - No-terminales
    - Terminales

# Construcción de un analizador predictivo (II)

---

```
procedure A;  
begin  
  if lookahead pertenece a PRIMERO( $\alpha$ ) then  
    -- analizar  $\alpha$   
  else if lookahead pertenece a PRIMERO( $\beta$ ) then  
    -- analizar  $\beta$   
  else  
    -- error  
  end if;  
end A;
```

# Ejemplo

---

tipo  $\rightarrow$  tipo\_simple

|  $\uparrow$ id

| array [ tipo\_simple ] of tipo

tipo\_simple  $\rightarrow$  integer

| char

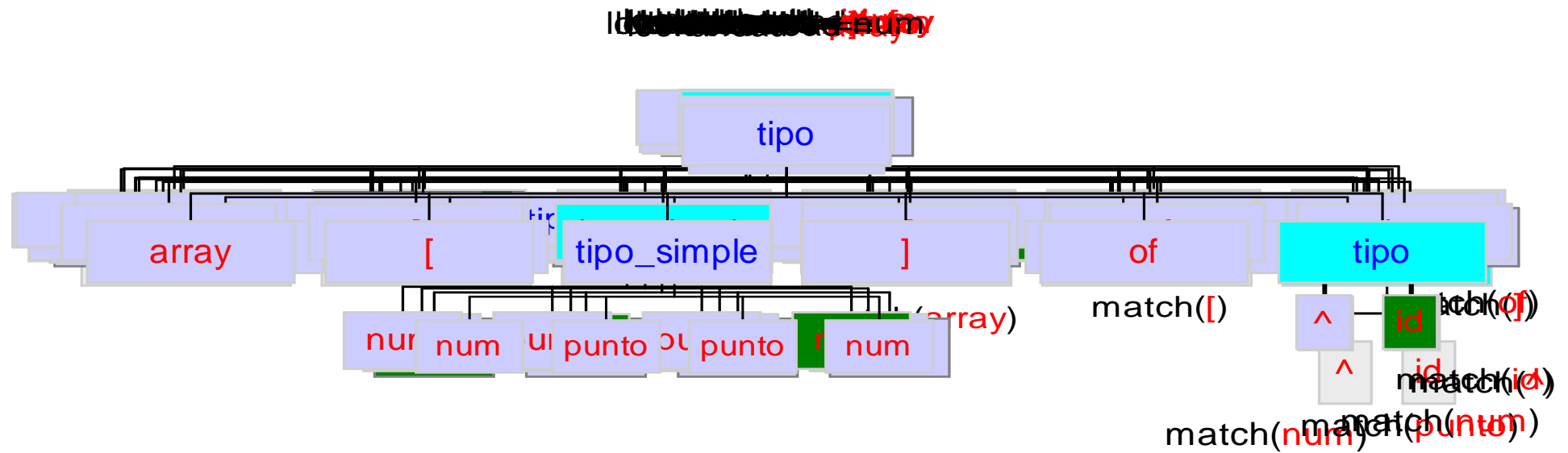
| num punto punto num

# Implementación del ejemplo

---

```
procedure match(token: tipo_token);  
begin  
    if lookahead = token  
        then lookahead := siguiente_token()  
    else error  
end;
```

# array [1..10] of ^ persona



# Implementación del ejemplo

---

```
procedure tipo;  
begin  
  if lookahead en { integer char num }  
    then tipo_simple  
  else if lookahead en { ↑ }  
    then match(↑); match(id);  
  else if lookahead en { array }  
    then match(array); match('[');  
          tipo_simple; match(']');  
          match(of); tipo;  
  else error  
end;
```

# Implementación del ejemplo

---

```
procedure tipo_simple;  
begin  
    if lookahead en { integer }  
        then    match(integer)  
    else if lookahead en { char }  
        then    match(char);  
    else if lookahead en { num }  
        then    match(num);  
                match(punto); match(punto);  
                match(num)  
    else error  
end;
```

# Transformación de una gramática en predictiva

---

Acciones a realizar en la gramática para conseguir un analizador predictivo:

1. Eliminar la ambigüedad
2. Eliminar la recursividad a izquierdas
3. Factorizar



# Recursividad a izquierdas

---

- Existe recursividad a izquierdas (inmediata) si en la gramática hay alguna regla con la forma  $A \rightarrow A \alpha$
- La presencia de recursividad a izdas. inmediata en una gramática impide la construcción de un analizador descendente predictivo

# Recursividad a izquierdas (II)

---

- $A \rightarrow A\alpha \mid \beta$  donde  $L(A)=L(\beta) L(\alpha)^n$   $n \geq 0$

Se transforma en:

- $A \rightarrow \beta A'$   
 $A' \rightarrow \xi \mid \alpha A'$  donde  $A, A' \in N \wedge \alpha, \beta \in (T \cup N)^*$

# Factorización

---

Dado  $A \rightarrow \alpha \beta \mid \alpha \gamma$

Podemos transformarlo en:

$A \rightarrow \alpha C$                       siendo  $A, C \in N$

$C \rightarrow \beta \mid \gamma$                        $\alpha, \beta, \gamma \in (T \cup N)^*$

# Otra vez las expresiones

---

$$E \rightarrow T E'$$
$$E' \rightarrow + T E'$$
$$| - T E'$$
$$| \xi$$
$$T \rightarrow F T'$$
$$T' \rightarrow * F T'$$
$$| / F T'$$
$$| \xi$$
$$F \rightarrow \text{entero}$$
$$| \text{identificador}$$

# Ejemplo de analizador predictivo (I)

---

```
procedure E is
begin
  if -- lookahead es un entero o un id. -- then
    T;
    E_PRIMA;
  else
    -- tratar error
  end if;
end E;
```

# Ejemplo de analizador predictivo (II)

---

```
procedure E_PRIMA is
begin
  if -- lookahead es un {+} -- then
    -- empareja el +
    T; E_PRIMA;
  else if -- lookahead es un {-} -- then
    -- empareja el -
    T; E_PRIMA;
  else -- no hacer nada
  end if;
end E_PRIMA;
```

## 3.5 Traducción de los principales constructores de los L.P.

---

- Expresiones aritméticas
- Expresiones booleanas
- Sentencias condicionales
- Bucles

# Traducción de las expresiones aritméticas

---

- Traductor para expresiones aritméticas en que los enteros pueden representarse en su versión habitual o en notación romana
- Ejemplo: **V - III - 2** se traduciría en  
     $t1 := 5 - 3;$   
     $t2 := t1 - 2;$



# Ejemplo de transformación de E.T.D.S.(calculadora)

---

(1)  $E \rightarrow E - \text{entero}$        $\{E.\text{valor} := E_1.\text{valor} - \text{entero}.\text{valor}\}$   
        $| \text{entero}$                        $\{E.\text{valor} := \text{entero}.\text{valor}\}$

(1)  $E \rightarrow \text{entero}$        $\{E'.\text{hvalor} := \text{entero}.\text{valor}\}$   
        $E'$                        $\{E.\text{valor} := E'.\text{valor}\}$

(2)  $E' \rightarrow - \text{entero}$        $\{E'_1.\text{hvalor} := E'.\text{hvalor} - \text{entero}.\text{valor}\}$   
        $E'$                        $\{E'.\text{valor} := E'_1.\text{valor}\}$   
        $| \xi$                        $\{E'.\text{valor} := E'.\text{hvalor}\}$



# Atributos y ETDS

---

- Atributos son locales:
  - » Locales a la producción = locales al subárbol
- Sintetizados:
  - » Recogen su valor de los atributos de sus nodos hijos.
- Heredados:
  - » El valor viene del padre y/o de los hermanos a su izquierda.

# E.T.D.S.

## (traducción de expresiones)

---

(1)  $I \rightarrow \mathbf{id} := E;$      $\{\text{añadir\_inst}(\mathbf{id.nombre} || ' := ' || E.nombre)\}$

(2)  $E \rightarrow E - E$      $\{E.nombre := \text{obtener\_nuevo\_id};$   
     $\text{añadir\_inst}(E.nomb || ' := ' || E1.nomb || '-' || E2.nomb)\}$

    | **entero**     $\{E.nombre := \mathbf{entero.nombre}\}$

    | **id**     $\{E.nombre := \mathbf{id.nombre}\}$

?



# E.T.D.S. para E

## (trad. de exp. – no ambigua)

---

(2a) $E \rightarrow E - F$	$\{E.\text{nombre} := \text{obtener\_nuevo\_id};$ $\text{añadir\_inst}(E.\text{nomb}    ' := '    E_1.\text{nomb}    ' - '    F.\text{nomb})\}$
$  F$	$\{E.\text{nombre} := F.\text{nombre}\}$

(2b) $F \rightarrow \text{entero}$	$\{F.\text{nombre} := \text{entero.nombre}\}$
$  \text{id}$	$\{F.\text{nombre} := \text{id.nombre}\}$

?



# E.T.D.S. para E

## (trad. de exp. – descendente pred.)

---

(2a')  $E \rightarrow F \quad \{E'.hnombre := F.nombre\}$   
 $E' \quad \{E.nombre := E'.nombre\}$

(2a'')  $E' \rightarrow - F \quad \{E'_1.hnombre := obtener\_nuevo\_id;$   
 $\quad \quad \quad \text{añadir\_inst}(E'_1.hn||':='||E'.hn||'-'|| F.nombre)\}$   
 $E' \quad \{E'.nombre := E'_1.nombre\}$   
 $| \xi \quad \{E'.nombre := E'.hnombre\}$

# E.T.D.S. para E

## (trad. de exp. – descendente pred.)

---

(1)	$I \rightarrow$	<b>id</b> := E;	$\{\text{añadir\_inst}(\text{id.nombre}    ' := '    \text{E.nombre})\}$
(2a')	$E \rightarrow$	F	$\{E'.\text{hnombre} := F.\text{nombre}\}$
		E'	$\{E.\text{nombre} := E'.\text{nombre}\}$
(2a'')	$E' \rightarrow$	– F	$\{E'_1.\text{hnombre} := \text{obtener\_nuevo\_id};$ $\text{añadir\_inst}(E'_1.\text{hnombre}    ' := '    E'.\text{hnombre}    -   $ $F.\text{nombre})\}$
		E'	$\{E'.\text{nombre} := E'_1.\text{nombre}\}$
		$\xi$	$\{E'.\text{nombre} := E'.\text{hnombre}\}$
(2b)	$F \rightarrow$	<b>entero</b>	$\{F.\text{nombre} := \text{entero.nombre}\}$
		<b>id</b>	$\{F.\text{nombre} := \text{id.nombre}\}$

# E.T.D.S para E

---

- Abstracciones funcionales:
  - » Añadir\_inst
  - » Obtener\_nuevo\_id
- Atributos
  - » Heredados
    - E'.hn (hnombre)
  - » Sintetizados
    - E.nombre
    - F.nombre
    - E'.nombre

# Implementación de los atributos

---

- Para cada atributo de un no terminal se coloca un parámetro en su procedimiento:
  - Atributo sintetizado: out
  - Atributo heredado: in
- Se necesita una var. local para cada atributo de los símbolos de la parte dcha. de las regla de producción del no terminal.



# Ejemplo de un traductor predictivo

---

```
procedure E (E_nombre: out string) is  
    E_prima_hnombre, F_nombre, E_prima_nombre:string;  
begin  
    F (F_nombre);  
    E_prima_hnombre:= F_nombre;  
    E_prima(E_prima_hnombre, E_prima_nombre);  
    E_nombre := E_prima_nombre;  
end E;
```

# Ejercicio (I)

---

- Especificar la traducción de una declaración de la forma:

**var** *lista\_de\_ident* : **entero** o **real**

a una lista de declaraciones simples con la forma:

*ident* : **entero** (o *real*)

## Ejercicio (II)

---

- Especificar la traducción de una declaración de la forma:

**entero o real** *lista\_de\_ident* ;

a una lista de declaraciones simples con la forma:

*ident* : **entero** (o *real*)

# Ejercicio (III)

---

- Especificar la traducción de un programa formado por una secuencia de instrucciones de asignación de la forma *id*  $:=$  *expresion* en una secuencia de instrucciones de asignación sencillas (máximo tres operandos).

# Ejercicio (IV)

---

- Especificar la traducción de las siguientes sentencias:
  - » Loop
  - » expresión booleana
  - » If
  - » while

# Soluciones (I.1)

---

Declaraciones → **var** lista\_ident : tipo ;

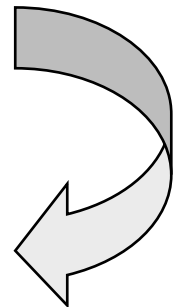
tipo → **entero** | **real**

lista\_ident → lista\_ident , **ident**  
| **ident**

lista\_ident → **ident** resto\_lista

resto\_lista → , **ident** resto\_lista

|  $\xi$



# Soluciones (I.2)

---

Declaraciones  $\rightarrow$  **var** lista\_ident : tipo ;

{añadir\_declaraciones(lista\_ident.Inom, *tipo.clase*)}

tipo  $\rightarrow$  **entero**      {tipo.clase := “entero”}

| **real**                {tipo.clase := “real”}

*añadir\_declaraciones*: **Para cada** nom **en** lista\_ident.Inom  
**empezando por el primero y acabando por el último hacer:**

añadir\_inst(nom||:||tipo.clase)

# Soluciones (I.3)

---

lista\_ident  $\rightarrow$  **ident** resto\_lista

{lista\_ident.Inom := añadir(resto\_lista.Inom,**ident**.nom)}

resto\_lista  $\rightarrow$  , **ident** resto\_lista

{resto\_lista.Inom := añadir(resto\_lista1.Inom,**ident**.nom)}

|  $\xi$  {resto\_lista.Inom := lista\_vacia() }

*añadir*: añade un elemento a la lista por el principio de ésta



# Soluciones (II.1)

---

Declaraciones →

tipo {lista\_ident.htipo := tipo.clase} lista\_ident ;

lista\_ident → **ident**

{añadir\_inst(ident.nombre || ':' || lista\_ident.htipo);

resto\_lista.htipo := lista\_ident.htipo;}

resto\_lista

# Soluciones (II.2)

---

resto\_lista →

, **ident** {añadir\_inst(ident.nombre || ':' ||  
                  resto\_lista.htipo);

          resto\_lista1.htipo := resto\_lista.htipo}

resto\_lista

| ξ

# Solución (IV.1)

## Expresiones booleanas. Traducción con salto

---

$E \rightarrow E \text{ oprel } E$

{E.TRUE := INILISTA(OBTENREF);

E.FALSE := INILISTA(OBTENREF+1);

AÑAD\_INST('if' || E<sub>1</sub>.NOMBRE || oprel.tipo || E<sub>2</sub>.NOMBRE ||  
          'goto' \_);

AÑAD\_INST ('goto' \_)}

# Solución (IV.2)

## Instrucciones condicionales Traducción con saltos

---

(1)  $S \rightarrow \text{if } E \text{ then } M \text{ } S \text{ } N$   
 $\text{else } M \text{ } S \text{ } M$

{COMPLETAR (E.TRUE,  $M_1$ .REF);  
COMPLETAR (E.FALSE,  $M_2$ .REF);  
COMPLETAR (N.NEXT,  $M_3$ .REF);}

(2)  $N \rightarrow \xi$

{N.NEXT := INILISTA(OBTENREF);  
AÑAD\_INST ('goto' \_)}

# Solución (IV.3)

## Instrucciones condicionales

---

(3)  $S \rightarrow \text{if } E \text{ then } M \text{ S } M$

{COMPLETAR(E.TRUE,  $M_1$ .REF);  
COMPLETAR(E.FALSE,  $M_2$ .REF);}

# Solución (IV.4)

## Instrucciones repetitivas

---

(4)  $S \rightarrow \mathbf{while} \ M \ E \ \mathbf{do} \ M \ S \ M($   
    {COMPLETAR(E.TRUE,  $M_2$ .REF);  
    COMPLETAR(E.FALSE,  $M_3$ .REF+1);  
    AÑAD\_INST ('goto'  $M_1$ .REF)}

## 3.6 Análisis semántico

---

- Restricciones semánticas
- Validación estática y dinámica
- Especificación de un validador de tipos
- Equivalencia y conversión de tipos
- Funciones y operadores sobrecargados

# Restricciones semánticas

---

- Comprobaciones de tipos
- Comprobaciones de *flujo de control*
- Comprobaciones de declaración y de declaración única
- Comprobaciones relacionadas con nombres



# Validación estática y dinámica (I)

---

- La validación estática se realiza durante la **compilación** del programa
- La validación dinámica se realiza durante la **ejecución** del programa

El código que realiza la validación dinámica es generado por el compilador como parte de la traducción

# Validación estática y dinámica (II)

---

- La caracterización de un lenguaje de programación incluye la definición de qué se debe comprobar en tiempo de compilación y qué se debe comprobar en tiempo de ejecución

# Especificación de un validador de tipos

---

- Caracterizar desde el punto de vista de los tipos de datos las restricciones del lenguaje (sistema de tipos)
- Especificarlas de forma que se oriente la construcción de la parte del compilador que valida las restricciones
  - » tratamiento de errores

# Sistemas de Tipos

---

- Restricciones de los tipos de los operandos según el operador
- Definición del tipo del resultado de una operación
- Definición de si las restricciones se comprueban estática o dinámicamente
  - » sistemas de tipos “seguros” o fuertemente “tipados”

# Ejemplo de un validador de tipos

---

Programa → Declaraciones ; Expresion  
Declaraciones → Declaraciones ; Declaraciones  
Declaraciones → **id** : Tipo  
Tipo → **char**  
Tipo → **integer**  
Tipo → **array** [ **entero** ] of Tipo  
Tipo → **access** Tipo

# ¿Y las expresiones?

---

Expresion  $\rightarrow$  **literal**

Expresion  $\rightarrow$  **entero**

Expresion  $\rightarrow$  **id**

Expresion  $\rightarrow$  Expresion **mod** Expresion

Expresion  $\rightarrow$  Expresion [ Expresion ]

Expresion  $\rightarrow$  Expresion . **all**

# E.T.D.S que especifica la validación (I)

---

Programa  $\rightarrow$  Declaraciones ; Expresion

{Si Expresion.tipo = 'error' entonces  
tratar\_error}

Declaraciones  $\rightarrow$  Declaraciones ; Declaraciones

Declaraciones  $\rightarrow$

**id** : Tipo      {añadir\_tipo(id.nombre, Tipo.tipo)}

Tipo  $\rightarrow$  **char**    {Tipo.tipo := 'caracter'}

# E.T.D.S (II)

---

Tipo  $\rightarrow$  **integer**

{Tipo.tipo := 'entero'}

Tipo  $\rightarrow$  **array** [ **entero** ] of Tipo

{Tipo.tipo := array(entero.val, Tipo<sub>1</sub>.tipo)}

Tipo  $\rightarrow$  **access** Tipo

{Tipo.tipo := apuntador(Tipo<sub>1</sub>.tipo)}



# E.T.D.S. (III)

---

Expresion  $\rightarrow$  **literal**

{Expresion.tipo := 'caracter'}

Expresion  $\rightarrow$  **entero**

{Expresion.tipo := 'entero'}

Expresion  $\rightarrow$  **id**

{Expresion.tipo := obtener\_tipo(id.nombre)}

## E.T.D.S. (IV)

---

Expresion  $\rightarrow$  Expresion **mod** Expresion

{Si Expresion<sub>1</sub>.tipo = 'entero' y  
Expresion<sub>2</sub>.tipo = 'entero'

entonces Expresion.tipo := 'entero'

si no Expresion.tipo := 'error'}

# E.T.D.S. (V)

---

Expresion  $\rightarrow$  Expresion [ Expresion ]

{Si Expresion<sub>2</sub>.tipo = 'entero' y  
Expresion<sub>1</sub>.tipo = array(n,t)

entonces Expresion.tipo := t

si no Expresion.tipo := 'error'}

# E.T.D.S. (VI)

---

Expresion  $\rightarrow$  Expresion . **all**

{Si Expresion<sub>1</sub>.tipo = apuntador(t)

entonces Expresion.tipo := t

si no Expresion.tipo := 'error'}

# E.T.D.S. (abstracciones)

---

- añadir\_tipo (identificador,tipo)

Añade el *identificador* a la tabla de símbolos y le asocia el *tipo* de entrada

Si el *identificador* ya había sido declarado le asocia el tipo *error*

- obtener\_tipo (identificador)

Si el *identificador* ya está en la tabla de símbolos devuelve el tipo que tenga asociado, si no devuelve como tipo *error*

# Ejercicios

---

- La especificación anterior se ha construido sobre una gramática ambigua. Identifica qué problemas se plantean y modifica la especificación para resolverlos
- Modifica la especificación para que en vez de un único aviso de error al final, se den tantos avisos como errores semánticos se detecten

# Equivalencia y conversión de tipos

---

- Dentro de la especificación de tipos de un lenguaje se encuadra la definición de tipos **equivalentes**
- Es habitual que durante la traducción se generen instrucciones que realizan conversión de tipos

# Conversión (E.T.D.S.)

---

(1)  $A \rightarrow id := E$

{AÑAD\_INST(id.NOMBRE || ':= ' ||  
E.NOMBRE)}

(2)  $E \rightarrow E + E$

{E.NOMBRE := nuevo\_ident();  
AÑAD\_INST (E.NOMBRE || ':= ' ||  
E<sub>1</sub>.NOMBRE || '+' || E<sub>2</sub>.NOMBRE)}

?





# Conversión (E.T.D.S.)

---

(2)  $E \rightarrow E + E$

{temp := nuevo\_ident();

if  $E_1.TIPO=INTEGER$  and  $E_2.TIPO=INTEGER$  then

    AÑAD\_INST (temp || ':= ' ||  $E_1.NOMBRE$  || "int\_sum" ||  
                     $E_2.NOMBRE$ );

$E.TIPO := INTEGER$ ;

else if  $E_1.TIPO=REAL$  and  $E_2.TIPO=REAL$  then

    AÑAD\_INST (temp || ':= ' ||  $E_1.NOMBRE$  || "real\_sum"  
                    ||  $E_2.NOMBRE$ );

$E.TIPO := REAL$

## E.T.D.S (II)

---

```
else if  $E_1$ .TIPO=INTEGER /*  $E_2$ .TIPO=REAL */ then  
    o_temp := nuevo_ident();  
    AÑAD_INST (o_temp || ':=' || "intareal" ||  
         $E_1$ .NOMBRE);  
    AÑAD_INST (temp || ':=' || o_temp || "real_sum" ||  
         $E_2$ .NOMBRE);  
    E.TIPO := REAL
```

## E.T.D.S. (III)

---

```
else /*  $E_1.TIPO=REAL$  y  $E_2.TIPO = INTEGER$ */ o_temp :=  
    nuevo_ident();  
    AÑAD_INST (o_temp || ' := ' || "intareal" ||  $E_2.NOMBRE$ );  
    AÑAD_INST (temp || ' := ' ||  $E_1.NOMBRE$  || "real_sum" ||  
                o_temp);  
  
    E.TIPO := REAL  
end if;  
E.NOMBRE := temp  
}
```

# Funciones y operadores sobrecargados

---

- Sobrecarga sintáctica y semántica
- Enlace dinámico y enlace estático

## 3.7 La Tabla de Símbolos

---

- Contenido y manejo de la Tabla de Símbolos
- Representación de la Tabla de Símbolos
- El alcance de la información

# Contenido y manejo de la Tabla de Símbolos (I)

---

- Es una estructura de datos donde el compilador almacena toda la información que necesita sobre los objetos (constantes, tipos, variables, procedimientos, funciones,...) que aparecen, identificados por un nombre, en los programas
- Cada entrada de la Tabla es un par (nombre,información)

# Contenido y manejo de la Tabla de Símbolos (II)

---

- La Tabla de Símbolos es utilizada prácticamente en todas las fases del compilador
- Existen distintas alternativas a la hora de definir las funcionalidades de la Tabla de Símbolos, escoger entre las diversas alternativas es una decisión a adoptar por el diseñador del compilador

# Funcionalidades de la Tabla de Símbolos

---

- Inicializar la Tabla de Símbolos
- Añadir un identificador a la Tabla
- Determinar si un identificador se encuentra en la Tabla
- Añadir información asociada a un identificador
- Obtener la información asociada a un identificador
- Borrar un identificador o un grupo de identificadores de la Tabla



# Tipo de información que contiene la T. de S.

---

- La secuencia de caracteres que corresponden a un identificador
- Atributos de un identificador: Clase, tipo...
- Parámetros asociados:
  - » Tablas: Número de dimensiones y rango de valores de los índices...
  - » Registros: Nombres y tipos de los campos del registro
  - » Procedimiento: Número de parámetros y su tipo...
  - » Referencias a la dirección de memoria que le corresponde

# Contenido y manejo de la Tabla de Símbolos

---

- La información se añade y consulta en distintos (muchos) momentos
- Hay que considerar el problema que conlleva representar informaciones que pueden tener un tamaño variable (dimensiones de un array...)
- Posibilidad de reutilizar parte del espacio utilizado por la T. de S.
- Para procedimientos y funciones
  - » registro de activación

# Representación de la Tabla de Símbolos

---

- Listas
- Árboles binarios de búsqueda
- Tablas Hash

# Lista

---

- La más sencilla
- La de peor rendimiento
- Implementación: estática o dinámica
- Mejoras posibles: ordenación alfanumérica, con respecto al acceso más reciente,...

# Arboles binarios de búsqueda

---

- Si el árbol binario de búsqueda es equilibrado el rendimiento será mejor que el de una lista, siendo la solución algo más compleja
- Implementación: Estática o dinámica
- Observación: si el árbol es totalmente *degenerado* el rendimiento es equivalente al de la lista

# Tablas HASH

---

- Muchas opciones de diseño
- Mejor rendimiento a costa de una mayor complicación de diseño e implementación
- Implementación estática o semidinámica
- La eficiencia del método depende de la habilidad en la selección de la función hash y en el tratamiento de sinónimos

# Ideas generales sobre Tablas Hash (I)

---

- Adecuadas para estructuras de datos de tipo diccionario
- Un función hash se define:
  - » Fhash: Claves -> Direcciones
- Hashing cerrado y hashing abierto
- Problema clave: Buscar una función que distribuya uniformemente las claves entre las referencias a la estructura de datos (direcciones)

# Tablas Hash (II)

---

- Función más sencilla (y clásica) para el acceso a un array de  $n$  elementos:
  - »  $H(c) = \text{ord}(c) \bmod n$
- Elementos sinónimos: Aquellos que teniendo claves distintas comparten el valor asociado por la función hash ( $H(c1)=H(c2)$ )



# Tablas Hash (III)

---

- Manejo de colisiones: búsqueda secuencial, funciones secundarias, enlazar sinónimos, reservar áreas especiales para sinónimos...
- Bibliografía: Algoritmos + Estructuras de Datos = Programas,...

# El alcance de la información

---

**program** ejemplo;

**var** aux, bloque : integer;

A

**procedure** P1 (**var** resul : integer);

**var** bloque : integer;

B

**begin**

**read**(bloque);

resul := bloque + aux;

B

**end;**

# Estado de la pila

---

A



1 (0)	→	ejemplo,aux,bloque
-------	---	--------------------

B

1 (0)	→	ejemplo,aux,bloque,P1
2 (1)	→	resul,bloque

# El alcance de la información (II)

---

```
procedure P3 (var resul : integer);  
var otra : integer;   
    function P31 (var arg : integer): boolean;  
    var aux : integer;   
    begin  
        read(aux);  
        P31 := (aux mod arg) = bloque;  
    end;
```

# Estado de la pila (II)

---

1 (0)	→	ejemplo,aux,bloque,P1,P3
3 (1)	→	resul,otra





C

1 (0)	→	ejemplo,aux,bloque,P1,P3
3 (1)	→	resul,otra,P3 1
4 (3)	→	arg, aux

D

# El alcance de la información (III)

---

```
begin 
    otra := bloque;
    resul := P31(otra); 
end;
begin 
    read(aux); bloque:= aux*2;
    P3(bloque); 
end;
```

# Estado de la pila (III)

---

E

1 (0)	→
3 (1)	→

ejemplo,aux,bloque,P1,P3
resul,otra,P3 1

F

1 (0)	→
-------	---

ejemplo,aux,bloque,P1,P3
--------------------------

## 3.8 Tipos de código intermedio

---

- Códigos de tres direcciones
- Máquinas abstractas con pila
  - » JVM
  - » p-code
- Notación postfija, árboles sintácticos y AST's



# La Máquina Virtual Java

---

- Desarrollo en conjunción con Java, pero susceptible de ser utilizado en la traducción de otros lenguajes
- Máximo de 256 instrucciones (1 byte para el código). Actualmente compuesto por 220 instrucciones.
- No "ortogonal"

# Estructura de la máquina virtual

---

- El registro **pc** (program counter)
- La pila (una por subproceso - *thread*)  
Frames (registros de activación)
- El heap
- Zona para el código

# Tipos de datos primitivos

---

- **byte** (1 byte)
- **short** (2 bytes)
- **int** (4 bytes)
- **long** (8 bytes)
- **char** (2 bytes, versión Unicode)
- **float** (4 bytes)
- **double** (8 bytes)
- **returnAdress** (que no es un tipo de Java)
- **reference** (incluido el valor especial null)

# Tipos de instrucciones (I)

---

- **De carga y almacenamiento entre la pila de operandos**
  - » Tload (carga), Tstore (almacenamiento),...
- **Instrucciones aritméticas**
  - » Suma, resta, producto, división, resto, incremento,...
- **"Instrucciones aritméticas"**
  - » desplazamiento, or, and, xor
- **Instrucciones de conversión de tipos**
  - » i2l (entero a entero largo - int to long)

# Tipos de instrucciones (II)

---

- **Creación y manipulación de objetos**
  - » crear instancias de clases y de arrays
- **Invocación de métodos y return's**
  - » invoke y Treturn
- **Instrucciones de salto**
  - » goto (incondicional)
  - » if<cond>, if\_icmp<cond> (condicional)
- **Salto con varias opciones:** tableswitch, ...

# Soporte de tipos (ejemplo)

<b>código</b>	byte	short	int	long	float	double
Tipush	X	X				
Tconst			X	X	X	X
Tload			X	X	X	X
Tstore			X	X	X	X
Tinc			X			
Taload	X	X	X	X	X	X
Tastore	X	X	X	X	X	X
Tadd			X	X	X	X
...						
i2T	X	X		X	X	X
l2T			X		X	X
...						
Tcomp				X		
...						
if_TcmpOP			X			
...						

# Descripción de instrucciones (I)

---

- **if\_icmp<cond>**
- **Operación**
  - » Salto si la comparación entre enteros es cierta.
- **Formato**

if_icmp<cond>
branchbyte1
branchbyte2

# Descripción de instrucciones (II)

---

- **Formas**

- » if\_icmpeq, if\_icmpne, if\_icmplt, if\_icmpge, if\_icmpgt, if\_icmple

- **Pila**      **#, ..., valor2, valor1**

- ⇒ #, ...

- **Descripción** Tanto el valor1 como el valor2 deben ser de tipo int. Ambos son desempilados y comparados.



# Ejemplo

---

*The Java Virtual Machine Specification  
(Addison-Wesley). Pág. 341*

```
void spin() {  
    int i;  
    for (i = 0; i < 100; i++) {  
        ;    // cuerpo del ciclo vacío)  
    }  
}
```

# Ejemplo (traducción)

---

```
0 iconst_0 // Empila la constante entera 0
1 istore_1 // Del tope a la posición 1 (i=0)
2 goto 8    // Salto al cuerpo de inst. del for
5 iinc 1 1   // i++
8 iload_1   // Empila la variable i
9 bipush 100 // Empila la constante 100
11 if_icmplt 5 // Si (i < 100) salta a 5
14 return    // return void
```

# Código P

---

- Extendido en los compiladores de Pascal
- Compuesto por unas 130 instrucciones
- Tipos: dirección, boolean, carácter, entero, real y conjunto
- *Pascal Implementation, S. Pemberton*

# Algunas instrucciones (*p-code*)

---

<i>abi</i>	(i)	i	Valor absoluto
<i>adr</i>	(r,r)	r	Suma de reales
<i>chk</i>	(i)	c	Convierte a carácter
<i>csp</i>	(r,r)	r	Llamada a procedimiento
<i>equ</i>	(x,x)	b	Comparación de igualdad
<i>not</i>	(b)	b	Negación de un booleano
<i>sto</i>	sin efecto		Stop
<i>sto</i>	sin efecto		Error en la instrucción <i>case</i>

## 3.9 Extensión de la traducción de los principales constructores de los lenguajes de programación

---

- Declaraciones
- Instrucciones de asignación
- Expresiones booleanas
- Instrucciones compuestas
- Referencias a Tablas

# Declaraciones (I)

---

(1)  $P \rightarrow \{\text{despl}:=0\} D; S$

(2)  $D \rightarrow D;D$

(3)  $D \rightarrow \text{iddec} : T$

$\{\text{introduce}(\text{iddec.nombre}, T.\text{TIPO}, \text{despl});$   
 $\text{despl} := \text{despl} + T.\text{TAMAÑO};\}$

# Declaraciones (II)

---

(4)  $T \rightarrow \mathbf{integer}$

$\{T.TIPO := \text{entero}; T.TAMAÑO := 4\}$

(5)  $T \rightarrow \mathbf{real}$

$\{T.TIPO := \text{real}; T.TAMAÑO := 8\}$

(6)  $T \rightarrow \mathbf{array [ num ] of T}$

$\{T.TIPO := \text{array}(\mathbf{num.VAL}, T_1.TIPO);$   
 $T.TAMAÑO := T_1.TAMAÑO * \mathbf{num.VAL}\}$

# Declaraciones (III)

---

(7)  $T \rightarrow \mathbf{access} \ T$

$\{T.TIPO := \text{apuntador}(T_1.TIPO);$   
 $T.TAMAÑO := 4\}$

(8)  $\text{iddec} \rightarrow \mathbf{id}$

$\{\text{if DECLARADO}(\mathbf{id.NOMBRE})$   
 $\text{then ERROR};$   
 $\text{iddec.NOMBRE} := \mathbf{id.NOMBRE}\}$



# Instrucciones de asignación

---

**$S \rightarrow id := E$**

- Asignaciones con símbolos

{if incompatibles(obtener\_tipo(id.nombre),E.Tipo)  
then tratar\_error  
else añad\_inst(id.nombre|| ':='|| E.NOMBRE)}

- Asignaciones con direcciones

{if ... then tratar\_error  
else añad\_inst(obten\_dir(id.nombre)|| ':='|| E.dir)}

# Expresiones booleanas (I)

---

$E \rightarrow E \text{ or } E$

$E \rightarrow E \text{ and } E$

$E \rightarrow \text{not } E$

$E \rightarrow ( E )$

$E \rightarrow \text{id}$  *(cuando existe el tipo booleano)*

$E \rightarrow E \text{ oprel } E$  *(operador relacional)*

# Expresiones booleanas (II)

## Traducción numérica

---

- Traducción a representación numérica de los valores cierto/falso

$E \rightarrow E \text{ oprel } E$

```
{E.NOMBRE :=nuevo_ident();  
  AÑAD_INST('if' || E1.NOMBRE|| oprel.tipo  
    ||E2.NOMBRE||'goto' ||OBTENREF + 3);  
  AÑAD_INST(E.NOMBRE||':='||'0');  
  AÑAD_INST('goto'|| OBTENREF + 2);  
  AÑAD_INST(E.NOMBRE||':='||'1')}
```

# Expresiones booleanas (III)

## Traducción numérica

---

**$E \rightarrow E \text{ or } E$**

{E.NOMBRE :=nuevo\_ident();  
AÑAD\_INST(E.NOMBRE||':='||E<sub>1</sub>.NOMBRE|| 'or' ||  
E<sub>2</sub>.NOMBRE)}

**$E \rightarrow E \text{ or } E$**

{E.NOMBRE :=nuevo\_ident();  
AÑAD\_INST(E.NOMBRE||':='||E<sub>1</sub>.NOMBRE ||'+ ' ||  
E<sub>2</sub>.NOMBRE)}

# Expresiones booleanas (IV)

## Traducción con saltos

---

- Para manipular listas de referencias a instrucciones:
  1. **INILISTA (*i*)** crea una nueva lista conteniendo únicamente *i*, una referencia a una instrucción
  2. **UNIR(*I*<sub>1</sub>, *I*<sub>2</sub>)** dadas las listas *I*<sub>1</sub> y *I*<sub>2</sub> devuelve su unión
  3. **COMPLETAR(*I*,*etq*)** completa con la etiqueta *etq* las instrucciones referenciadas por la lista *I*

# Expresiones booleanas (V)

## Traducción con saltos

---

$M \rightarrow \xi$

{M.REF := OBTENREF}

$E \rightarrow E \text{ or } M E$

{COMPLETAR ( $E_1$ .FALSE, M.REF);

$E$ .TRUE := UNIR( $E_1$ .TRUE,  $E_2$ .TRUE);

$E$ .FALSE :=  $E_2$ .FALSE}

# Expresiones booleanas (VI)

## Traducción con saltos

---

**$E \rightarrow E \text{ and } M E$**

{COMPLETAR ( $E_1$ .TRUE, M.REF);  
 $E$ .TRUE :=  $E_2$ .TRUE;  
 $E$ .FALSE := UNIR( $E_1$ .FALSE,  $E_2$ .FALSE)}

# Expresiones booleanas (VII)

## Traducción con saltos

---

$E \rightarrow \text{not } E$

$\{E.\text{TRUE} := E_1.\text{FALSE};$   
 $E.\text{FALSE} := E_1.\text{TRUE}\}$

$E \rightarrow ( E )$

$\{E.\text{TRUE} := E_1.\text{TRUE};$   
 $E.\text{FALSE} := E_1.\text{FALSE}\}$



# Expresiones booleanas (VIII)

## Traducción con salto

---

$E \rightarrow E \text{ oprel } E$

{E.TRUE := INILISTA(OBTENREF);

E.FALSE := INILISTA(OBTENREF+1);

AÑAD\_INST('if' || E<sub>1</sub>.NOMBRE || oprel.tipo || E<sub>2</sub>.NOMBRE ||  
"goto" \_);

AÑAD\_INST ("goto" \_)}  
}

# Instrucciones de salto

---

- Manejar las etiquetas y las instrucciones goto

$S \rightarrow \text{etiqueta} : S$

$\text{etiqueta} \rightarrow \mathbf{id}$

$\{\text{GUARDA\_ETQ}(\text{id.NOMBRE}, \text{OBTENREF})\}$

- Y la instrucción de salto:

$S \rightarrow \mathbf{goto\ id}$

$\{e := \text{OBTEN\_DIR\_ETQ}(\text{id.NOMBRE});$

$\text{AÑAD\_INST}(\text{"goto"} \parallel e)\}$

# Referencias a Tablas

---

- Enfoque presentado:
- Expandir las referencias a elementos del array en el código intermedio
  - » permite realizar la optimización de accesos a elementos de arrays independientemente del lenguaje objeto

# Suposiciones de partida

---

- Para simplificar el problema de la traducción de referencias a arrays consideraremos que:
  - » Trabajamos con arrays de los que conocemos los límites en tiempo de compilación
  - » Cada elemento del array ocupa una palabra
  - » La máquina objeto tiene la memoria organizada por palabras

# Ejemplo

---

- A: array (10,20) of integer
- En memoria los elementos del array ocuparán posiciones de memoria consecutivas de la siguiente forma:
  - »  $A[1,1], A[1,2], \dots, A[10,19], A[10,20]$
- Sea  $a$  la dirección de la primera palabra del bloque de memoria correspondiente al array A. La posición del elemento  $A[e_i, e_j]$  será:
$$a + 20 * (\text{valor de } e_i - 1) + (\text{valor de } e_j - 1) =$$
$$(a - 21) + 20 * \text{valor de } e_i + \text{valor de } e_j$$

# Ejemplo - traducción (I)

---

El código necesario para obtener el valor de  $A[e_1, e_2]$  en una variable temporal será:

código para calcular  $e_1$  sobre  $t_1$

$t_2 := 20 * t_1$

código para calcular  $e_2$  sobre  $t_3$

$t_4 := t_2 + t_3$

$t := (a - 21)[t_4]$

$a$  dirección en que comienza el espacio asignado al array  $A$

# Ejemplo - traducción (II)

---

Si la dirección correspondiente a A sólo se conoce en tiempo de ejecución:

código para calcular e1 sobre t1

t2:=20\*t1

código para calcular e2 sobre t3

t4 := t2+t3

t5 := addr(A)

t6 := t5-21

t5 := t6 [t4]

# Generalización para $n$ dimensiones

---

- $A(d_1, d_2, \dots, d_n)$
- Número de componentes:  $d_1 \times d_2 \times \dots \times d_n$
- Acceso a  $A[e_1, \dots, e_n]$
- $a + (v_{e_1} - 1)d_2 \dots d_n + (v_{e_2} - 1)d_3 \dots d_n + \dots + (v_{e_n} - 1)$
- Páginas 496-500 de Compiladores,...