TEMA 2. Ampliación del lenguaje y ampliación de la máquina virtual:

- Instrucción compuesta, instrucciones de control y subrutinas

- Definición y construcción de tipos

1. Cuestiones previas
2. Restricciones contextuales de un lenguaje (tipos)
3. Construcción de un comprobador de tipos (Aho et al.)
- Asignar tipo a todas las construcciones del lenguaje
- Comprobar que se respetan las reglas de uso de los tipos
- Otros temas (*ver Aho et al.)*
4. ASPECTOS PRÁCTICOS:
1. Ampliación del lenguaje y construcción de las expresiones de tipos
2. Comprobador de tipos (el módulo de restricciones contextuales)
3. Generación de código (acceso a los componentes de los tipos)

- Procedimientos y funciones

Definición y construcción de tipos

1. Cuestiones previas:

- ¿Qué es un tipo?

Ideas informales:

- un tipo es un conjunto (dominio, rango) de elementos.
- un conjunto de propiedades (atributos) que determina si un elemento pertenece al tipo.
- definición de las operaciones válidas sobre los elementos del tipo y tipo del resultado
- definición de un tipo a partir de otros tipos (agregación y herencia)
- formas de representación (codificación) del conjunto de elementos de un tipo con los recursos de una máquina digital , *p.e.* codificación del tipo entero en dos *bytes* en complemento a dos, codificación del conjunto de los caracteres en uno o dos *bytes* (ASCII *o* UNICODE), el IEEE-754 para los reales, etc.

- De estas ideas las tres primeras forman parte del sistema de tipos de casi todos los lenguajes de programación modernos.
- La tercera y cuarta incluye elementos de tipos abstractos de datos y del paradigma orientado a objetos.
- La quinta está relacionada con cuestiones de más bajo nivel: la representación de los tipos primitivos de un lenguaje.

2. Restricciones contextuales de un lenguaje

(Aho, capítulo 6: Comprobación de tipos)

En un lenguaje de programación se introducen nombres para identificar los tipos y algunos de los elementos que pertenecen a un tipo.

Se introducen restricciones al uso de los nombres dependiendo del contexto en que éstos se utilizan:

- comprobaciones de unicidad de un nombre: en listas de declaración, en la instrucción *CASE*, etc.

- restricciones a la utilización de un nombre: no usar un nombre sin declararlo, limitar la utilización del contador de la instrucción *FOR*, etc.

El comprobador de tipos: Verifica las restricciones contextuales de un lenguaje. Un comprobador de tipos asegura que:

> *el tipo de cada construcción de un lenguaje de programación (categoría sintáctica) coincide con el previsto en su contexto.*

Para construir un comprobador de tipos es necesario:

1. Tener un sistema capaz de asignar un tipo (una **expresión de tipos**) a cada construcción (categoría sintáctica) del lenguaje.

2. Definir un conjunto de ecuaciones semánticas que garanticen que los tipos se utilizan correctamente.

Utilizaremos una gramática de atributos para los puntos 1 y 2

(hasta ahora habíamos considerado como inmediata la construcción de un comprobador de tipos para un lenguaje con dos, o tres, tipos básicos, y habíamos utilizado para su especificación una gramática de atributos).

Los tipos (expresiones de tipos) se incorporarán a la tabla de símbolos y no solo los utilizará el comprobador de tipos, también los utilizará el generador de código para obtener la información necesaria para la generación de código.

3. Construcción de un comprobador de tipos

(ver capítulo 6 de Aho et al.)

1. Asignar tipo a todas las construcciones (categorías sintácticas) del lenguaje

2. Comprobar que se respetan las reglas de uso de los tipos

3.1. Asignar tipo a todas las construcciones del lenguaje

En general en un lenguaje de programación distinguimos los siguientes tipos:

- básicos,
- construidos
- nombrados

La idea subyacente es distinguir entre tipos (conjuntos) definidos por el constructor del lenguaje y los definidos por el programador.

Expresiones de tipos

Un tipo se representa mediante una **expresión de tipos**.

Una expresión de tipos es una frase del lenguaje de los tipos:
el sub-lenguaje de las declaraciones

- *hasta ahora nos bastaba con un código entero (la mínima expresión de un lenguaje). En adelante esto no será suficiente: los tipos construidos y nombrados convierten el problema de representación de los tipos en un problema más complejo.*

- *inicialmente, la forma de representar un tipo básico es mediante un símbolo que da nombre al conjunto, p.e. integer*

- *en traducción dirigida por sintaxis, la forma de representar un tipo construido está basada en la estructura sintáctica de la frase que lo describe.*

  La pregunta ¿qué es una expresión de tipos? tiene el mismo sentido que la pregunta ¿qué es una expresión aritmética?

Las expresiones de tipos son frases del lenguaje de los tipos.

Otro problema es cómo codificamos o representamos estas frases para guardar la información asociada a las mismas en la tabla de símbolos, o como las procesamos "evaluamos" para obtener de ellas la información que nos hace falta.

El comprobador de tipos deberá ser capaz de construir (codificar) estas "**expresiones de tipos**" de acuerdo con las **declaraciones de tipos** y de utilizarlas después para realizar las comprobaciones de tipos pertinentes y extraer de ellas, "evaluarlas", la información necesaria para generar código.

3.1.1 Definición de las expresiones de tipos (según Aho et al.):

Una expresión de tipos es una frase del lenguaje (o sub-lenguaje) de las expresiones de tipos, que como tal, puede definirse de forma inductiva, o mediante una gramática.

En Aho et al. tenemos la siguiente definición inductiva (comparar esta definición con la definición inductiva de las expresiones aritméticas):

- **un tipo básico es una expresión de tipos.**

e.g. *integer, real, char, boolean*

Además se consideran dos tipos básicos especiales:

- *error_de_tipo*: para poder asignar tipo a una construcción del lenguaje de programación que tenga un error de tipo.

- *Vacío (OK)*: para asignar tipo a las construcciones del lenguaje que no lo tienen y son correctas (e.g. las instrucciones)

Los tipos enumerados y los subrangos también los consideraremos tipos básicos.

- **un tipo construido es una expresión de tipos**

Un tipo construido es el resultado de aplicar un constructor de tipos (un operador) a expresiones de tipos.

- **un nombre es una expresión de tipos**

Se puede dar un nombre a una expresión de tipos, por lo que el nombre de un tipo es también una expresión de tipos.
Nota:
La expresión de tipos se construye como una referencia (un operador referencia). Habrá que controlar si se producen autoreferencias (directas o indirectas).

3.1.2 Constructores de tipos (operadores de las expresiones de tipos)

- Producto cartesiano.
  Si T1 y T2 son expresiones de tipos: T1×T2 es una expresión de tipos.
  El operador "×" se considera asociativo por la izquierda.

- Punteros.
  Si T es una expresión de tipos: *pointer*(T) es una expresión de tipos

- Registros.
  Si n_i es el "nombre del campo$_i$" y T_i es una expresión de tipos:
  record(($n_1 \times T_1$) × ... × ($n_k \times T_k$)) es una expresión de tipos.

- *Arrays*.
  Si I es una expresión de tipos escalar: enumerado, subrango, etc. y T es una expresión de tipos: *array*(I, T) es una expresión de tipos.
  En un lenguaje como Pascal podríamos considerar I como un producto cartesiano de tipos escalares $I = I_1 \times ... \times I_n$.

- Funciones.
  Si T, T_1, ...T_n son expresiones de tipos $(T_1 \times .... \times T_n)$ -> T es una expresión de tipos [alternativamente $f(T_1 \times ... \times T_n)$: T]. La lista T_1...T_n puede estar vacía.

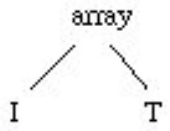
Nota:
Así se definen en el Aho las expresiones de tipos, con la misma técnica que se definieron las expresiones regulares (o las aritméticas).

En definitiva, definimos de forma inductiva el lenguaje (o sublenguaje): de los tipos.

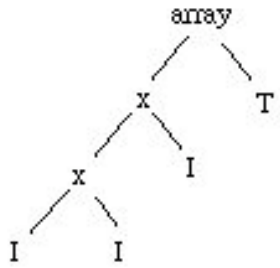
La pregunta ¿porqué no utilizar gramáticas para definir este lenguaje? quedará respondida después.

producto cartesiano (asociativo por la izquierda)

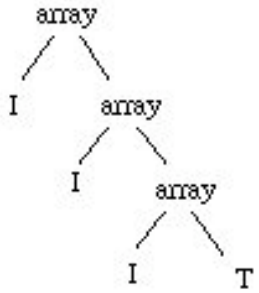
array



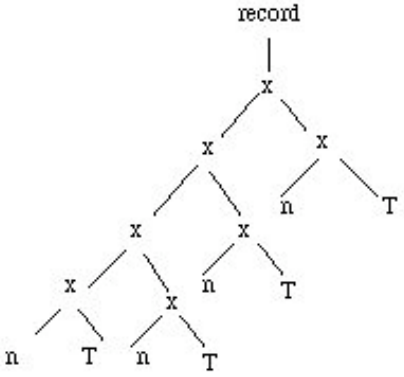
array multidimensional en Pascal



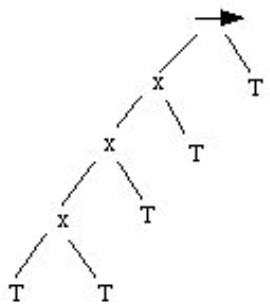
array multidimensional en C



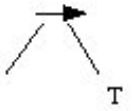
registro



función



función sin parámetros



Atención: un "identificador" puede representar el nombre de un tipo, el nombre de un campo, el nombre de una variable, el nombre de una constante, etc., por ello, introducimos un nuevo campo **"clase"** en la tabla de símbolos.

La **representación de las expresiones de tipos** estará basada en los árboles sintácticos:

- Los tipos básicos y los nombres son las hojas de los árboles que representan las expresiones de tipos.

- Los nodos son los constructores de tipos.

Otras posibles representaciones se pueden derivar de estos árboles utilizando cualquier técnica de codificación de árboles (ver el ejemplo 6.1 del Aho donde se utiliza una representación binaria)

3.2. Comprobar que se respetan las reglas de uso de los tipos

El comprobador de tipos puede actuar durante:

- la ejecución de un programa (dinámica),
- la compilación (estática),
- la compilación y la ejecución.

3.2.1 Comprobación dinámica.

- Durante la ejecución:

  En la práctica no es posible comprobar durante la compilación el tipo de todas las construcciones del lenguaje (e.g. comprobar que un término, en una expresión aritmética, no implica una división por cero, ó comprobar que un factor, que es un elemento de un *array*, está entre los límites de la declaración del *array*)

  e.g.

```
apila-dir(<numerador>)
apila-dir(<denominador>)
copia
apila(0)
igual
ir-falso(inst-div)
...<informar división por cero>
alto
etiqueta(inst-div)
dividir
...<resto de las instrucciones>
```

Otro caso interesante de especificación dinámica de tipos se utiliza en los lenguajes orientados a objetos cuando el nombre de un método (función) se liga dinámicamente con el código de un tipo más especializado.

3.2.2 Comprobación estática.

- Durante la compilación: En los lenguajes fuertemente tipados, el compilador incorpora un comprobador de tipos que garantiza que los programas (que no den error durante la comprobación) se ejecutarán sin errores de tipos.

- En compiladores de un solo paso, el comprobador de tipos actúa durante la compilación coordinado con el análisis y la generación de código intermedio.

3.2.3 Equivalencia de expresiones de tipos

¿cuándo son iguales, o equivalentes, dos expresiones de tipos?

El problema son las expresiones de tipos que tienen un nombre.

La cuestión clave es:

el nombre de un tipo en una expresión de tipos,
¿se representa a si mismo, o al tipo que nombra?

3.2.3.1 Equivalencia estructural vs. equivalencia por nombre

Dos tipos son estructuralmente equivalentes si se representan por la misma estructura, o por dos estructuras diferentes pero idénticas.

El **punto importante** es si en las expresiones de tipos los nombres se substituyen por las expresiones de tipos que nombran.

Los tipos nombrados permiten construir nuevos tipos en base a otros que hemos definido anteriormente.

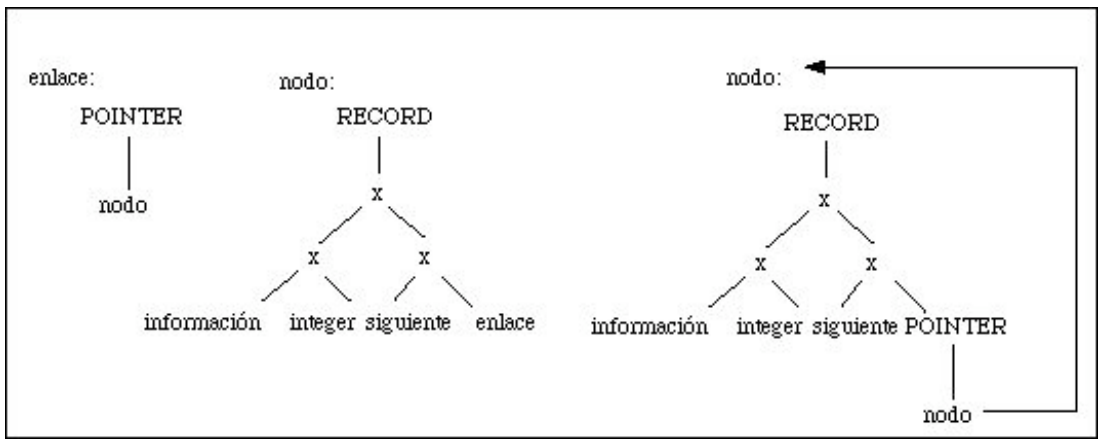
Un ejemplo típico son las estructuras dinámicas en C o Pascal.
p.e. con este fragmento de gramática

```
Dtipos --> TYPE  ident: Tipo

Tipo --> ident
         | integer
         | POINTER Tipo
         | RECORD Lcampos

Lcampos --> ident: Tipo
            | Lcampos ; ident: Tipo
```

podríamos declarar:

```
TYPE
enlace:  POINTER  nodo
TYPE
nodo: RECORD información: integer;
             siguiente: enlace
```

Estas construcciones posibilitan la recursión. En general los lenguajes no permiten la recursión directa pero si la recursión indirecta lo que implica que los árboles sintácticos se convierte en grafos con la consiguiente dificultad para comprobar la equivalencia estructural.



Ejemplos de casos especiales (en cuanto a equivalencia de tipos):

- En C los *arrays* son un caso especial (el comprobador de tipos no utiliza las límites del *array*) por ejemplo cuando un *array* se pasa como parámetro no se pasa la expresión de tipos completa.

- Pascal estándar no define cuando dos tipos son equivalentes. Además, algunas implementaciones de Pascal asocian implícitamente un nombre con cada declaración de variables.

- Expresiones de tipos definidas recursivamente (normalmente de forma indirecta).

  En el grafo que representa la expresión de tipos aparecerán bucles cuando substituyamos los nombres por el tipo que definen.

  La equivalencia estructural puede utilizarse aunque los grafos contengan ciclos pero como los algoritmos son mas costosos, C, por ejemplo, no utiliza la equivalencia estructural para los registros en cuya definición podemos utilizar de forma indirecta referencias a ellos mismos.

3.2.3.2 Algoritmo de equivalencia de tipos para estructuras sin ciclos (Aho).

Equivalencia estructural: "dos expresiones son, o bien el mismo tipo básico, o están formadas aplicando el mismo constructor a tipos estructuralmente equivalentes".

```
equiv(t1, t2):
```

- Si t1 y t2 son el mismo tipo básico
  equiv(t1, t2) = **true.**

- Si t1=**array**(a1, b1) y t2=**array**(a2, b2)
  equiv(t1, t2) = equiv(a1, a2) **and** equiv(b1,b2)

- Si t1=(a1 × b1) y t2=(a2 × b2)
  equiv(t1, t2) = equiv(a1, a2) **and** equiv(b1, b2)

- Si t1=**pointer**(a1) y t2=**pointer**(a2)
  equiv(t1, t2) = equiv(a1, a2)

- Si t1= a1 **->** b1 y t2= a2 **->** b2
  equiv(t1, t2) = equiv(a1, a2) **and** equiv(b1, b2)

discusión: en la práctica ¿qué carencias tiene este algoritmo?.

3.3 Otros temas (ver Aho et al.)

3.3.1 Conversiones de tipos.

Un tipo es un conjunto de datos y los elementos de cada conjunto suelen tener una codificación diferente en la máquina y diferentes instrucciones para operar con él.

Por ello cuando se utilizan tipos diferentes en una expresión, o instrucción, será necesario, caso de que sea semánticamente admisible, convertir todos los elementos a un mismo tipo.

Distinguimos dos tipos de conversiones: implícitas ó coerciones (realizadas automáticamente por el compilador) y explícitas "cast"(señaladas por el programador).

e.g. funciones `ord()` y `char()` en Pascal, o `cast()` en Java.

La conversión del tipo de las constantes si se hace durante la compilación puede mejorar mucho la eficiencia del código.

3.3.2 Sobrecarga de funciones y operadores

Símbolo sobrecargado: diferentes significados dependiendo del contexto.

Distinguimos dos casos:

a) se puede resolver a partir de los tipos de los operandos

b) el identificador de un operador, ó función, puede tener asociado en la tabla de símbolos un conjunto de tipos.

En la práctica una expresión debe tener un tipo único.

La técnica utilizada en Ada consiste en realizar un primer recorrido ascendente. En cada paso se obtiene un conjunto de tipos.

En la expresión completa (en el nodo raíz), este conjunto debe tener un único elemento.

En un segundo recorrido este tipo único se pasa como atributo heredado y se genera el código correspondiente.

3.3.3 Funciones polimórficas

Las funciones polimórficas permiten manipular estructuras de datos independientemente del tipo de sus elementos.

e.g. una función que determine la longitud de una lista independientemente del tipo de los elementos de la lista.

El problema más importante asociado con las funciones polimórficas es el de inferir el tipo de la función basándonos en su uso.

Es necesario introducir el concepto de *variable de tipos* para representar un tipo desconocido.

- utilización consistente del tipo de un identificador no declarado
- inferencia de tipos, inferir el tipo de una función a partir de su cuerpo o del modo en que se usa.

4. ASPECTOS PRÁCTICOS

1. **Ampliación del lenguaje y construcción de las expresiones de tipos**
2. **Comprobador de tipos (el módulo de restricciones contextuales)**
3. **Generación de código (acceso a los componentes de los tipos)**

Ampliación del lenguaje con tipos construidos y nombrados

Nuestro lenguaje imperativo mínimo tendrá, además de los tipos básicos, los tipos que se puedan construir con los siguientes **constructores (operadores)** de tipos.

- pointer, record, array, function, procedure

- el *producto cartesiano* como constructor de tipo auxiliar que se implementará de forma pragmática.

Representaremos las **expresiones de tipos** mediante estructuras derivadas de los **árboles sintácticos**

Mediante una **gramática de atributos** especificaremos como construir y manipular estas representaciones.

A las categorías sintácticas de la gramática de las expresiones de tipos les asociamos atributos que representan:

- punteros a los nodos de estos árboles.
- información (campos) asociada a los nodos de estos árboles.

Durante la compilación se pueden evaluar las expresiones de tipos para obtener información sobre el tipo:

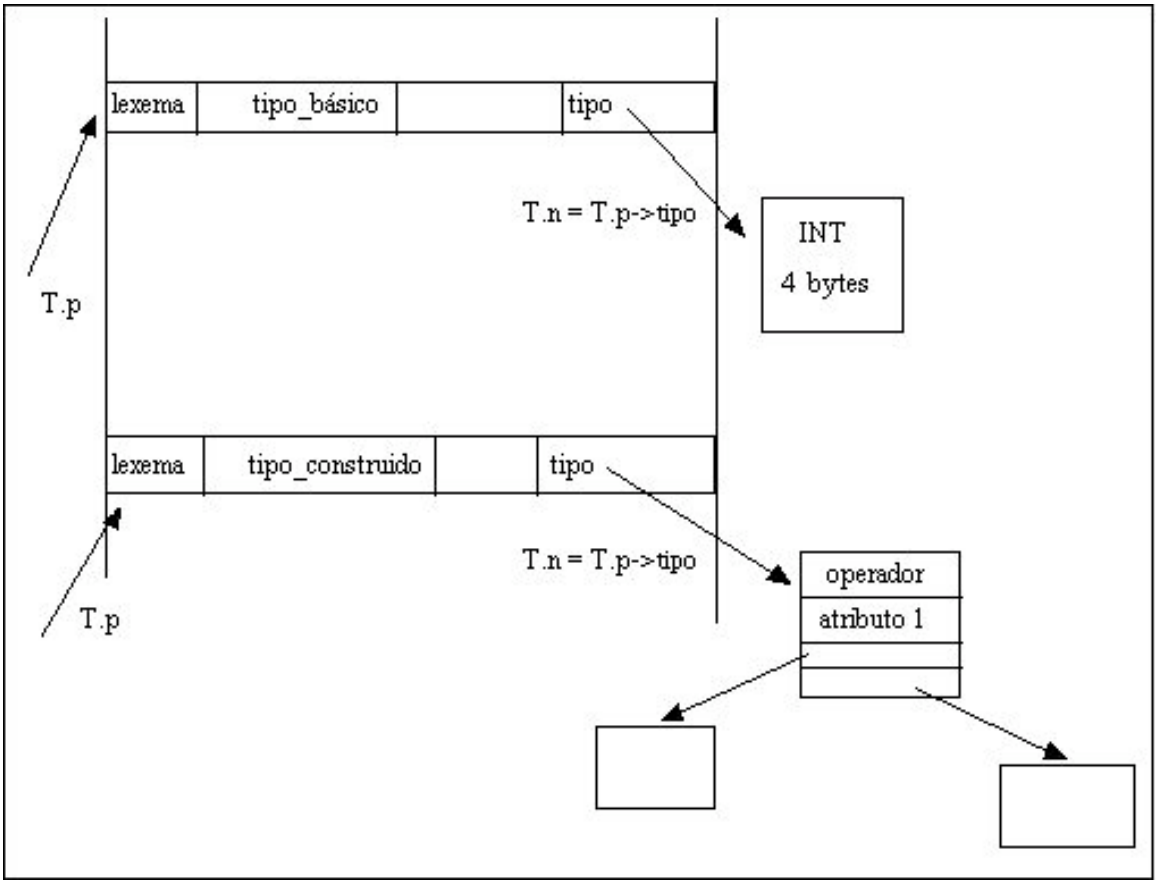
p.e.
- la cantidad de memoria necesaria para almacenar un elemento del tipo descrito
- el *offset,* o desplazamiento, desde una dirección base hasta cada componente del tipo.

1.- Ampliación del sub-lenguaje de las declaraciones y construcción de las expresiones de tipos

1.1.- Ejemplo de incorporación de tipos nombrados al lenguaje

Asociamos un nombre a una expresión de tipos (en forma similar a como asociábamos un nombre a una variable).

En este caso el nombre del tipo se incluirá en la tabla de símbolos y se le asignará la expresión de tipos
discusión: comparar con los lexemas en el ejemplo de Aho et al. y con las expresiones aritméticas del ejemplo de byte



La sintaxis siguiente nos permite incluir en nuestro lenguaje tipos nombrados.

Dtipo --> TYPE **ident**: Tipo
Dvar --> VAR ident: Tipo

Tipo --> integer | real | char | boolean
| ident
*(*un tipo puede ser una referencia nominal a otro tipo*)*

Fragmento de la gramática de atributos:

Atributos sintetizados de Tipo:

Tipo.p = puntero para acceder a la información sobre el símbolo en la tabla de símbolos

Tipo.n = puntero para acceder al nodo raíz de la expresión de tipos

Ecuaciones semánticas (se utiliza la notación de C para punteros a registros):

Tipo.p = busca(ident.lex)

if (Tipo.p<>*null*) *and*
(Tipo.p->clase = tipo_básico *or* Tipo.p->clase = tipo_construido)

then Tipo.n = Tipo.p->tipo
else error

Suponemos que **ident** representa el nombre de un tipo (para buscar información sobre un tipo básico utilizaremos la palabra reservada correspondiente.

discusión: ¿se permiten autoreferencias?

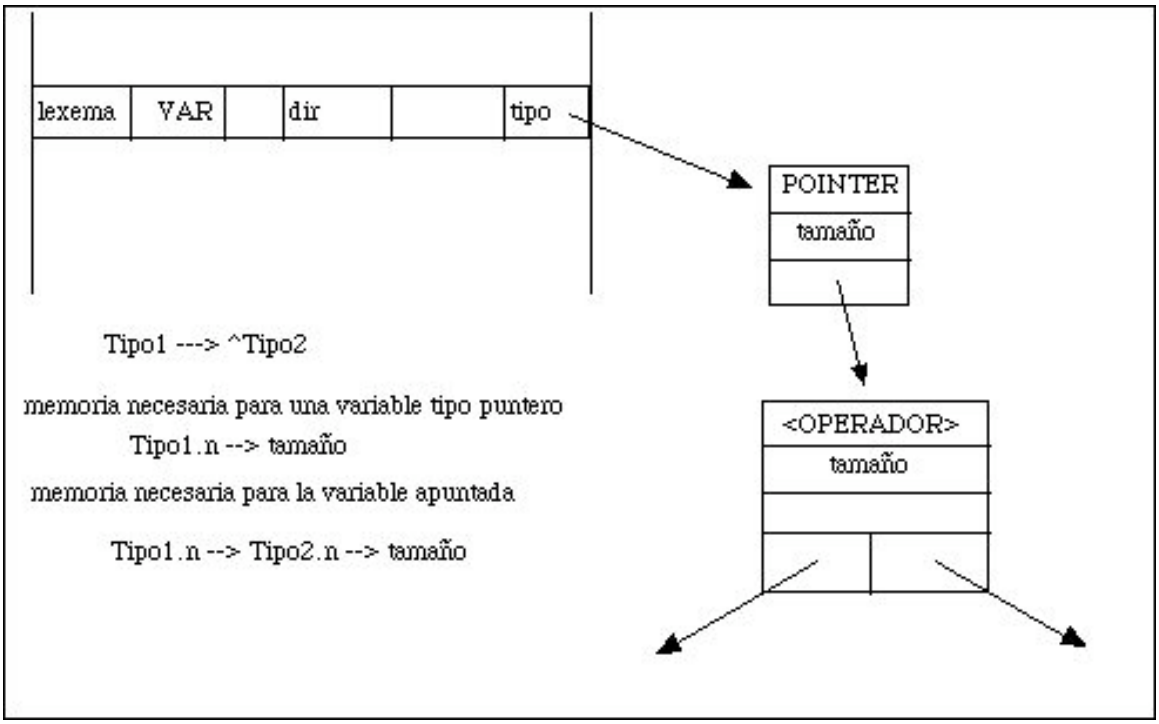
Buscamos el nombre del tipo en la tabla de símbolos y si ha sido definido previamente, o es un tipo básico, el campo tipo apuntará a un nodo con la información sobre el tipo

Suponemos que la tabla de símbolos se ha inicializado con todos los tipos básicos.

En ambos casos, el atributo sintetizado **Tipo.n** apunta a la definición del tipo (al nodo raíz de la expresión de tipos), es decir, al mismo sitio que el campo "tipo" de la entrada correspondiente en la tabla de símbolos.

ejemplos y posibles ampliaciones (rangos y enumerados)

1.2.- Ejemplo de la incorporación del tipo puntero (Pascal) y construcción de la expresión de tipos



Dtipos --> TYPE ident: Tipo
Dvar --> VAR ident: Tipo
Tipo --> integer | real | char | boolean | ident
Tipo --> ^ Tipo
$Tipo_1$.n = crea_nodo_pointer(*pointer*, $Tipo_2$.n)
$Tipo_1$.n ->tamaño = <bytes necesarios>
(- espacio que ocupa el puntero en el sistema)
(- comparar las funciones semánticas utilizadas con las utilizadas en las expresiones aritméticas)

La memoria necesaria para una variable depende de si esta está asociada al propio puntero o al tipo al que apunta el puntero.

en el primer caso accedemos a ese valor mediante la referencia:

$Tipo_1$.n->tamaño

en el segundo caso mediante la referencia:

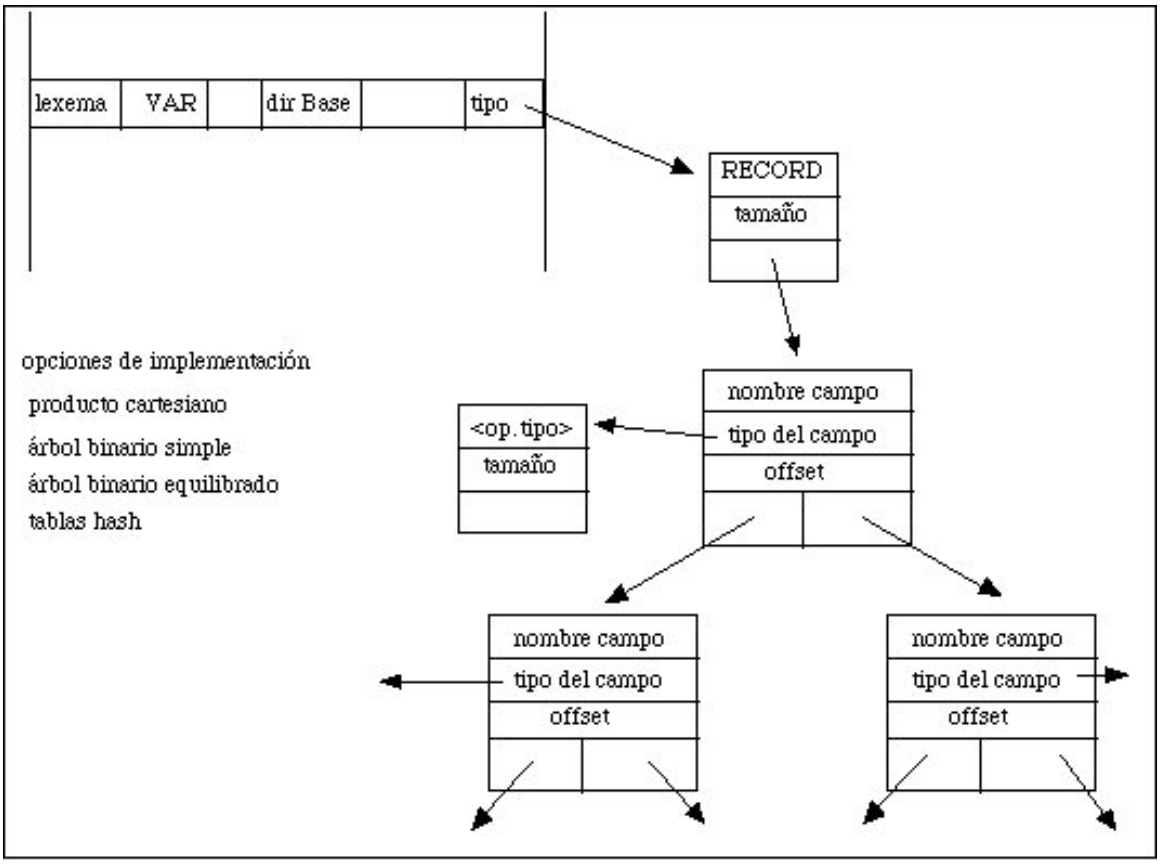
$Tipo_1$.n->$Tipo_2$.n->tamaño

1.3.- Ejemplo de la incorporación del tipo registro (Pascal) y construcción de la expresión de tipos

Dtipos --> TYPE ident: Tipo
Dvar --> VAR ident: Tipo
Tipo --> integer | real | char | boolean | ident
Tipo --> **^** Tipo

Tipo --> *record* Lcampos *end*
Lcampos --> Lident : Tipo
| Lcampos ; Lident : Tipo
Lident --> ident
| Lident , ident

- la categoría sintáctica Lcampos sirve para representar el producto cartesiano de los campos del registro
- las categoría sintáctica Lident y Tipo sirven para representar el producto cartesiano del nombre del campo y el tipo *(se puede simplificar y utilizar un solo identificador cada vez)*



Tipo --> ***record*** Lcampos ***end***

Tipo.n = crea_record_nodo(*record*, evaluar_offsets(Lcampos.n))
Tipo.n->tamaño = Lcampos.tamaño

- El atributo sintetizado **Lcampos.n** apunta a la raíz de la estructura que tiene toda la información sobre los campos del record (un árbol binario es una forma de representar el producto cartesiano de los campos que componen el *record* utilizada en algunos compiladores de Pascal)

- Cada nodo del árbol binario tiene la siguiente información:
  - lexema, nombre del campo
  - *offset* del campo
  - tipo del campo
  - hijo derecho e hijo izquierdo en el árbol binario

- El atributo **Lcampos.tamaño** sintetiza el valor de la memoria necesaria para almacenar el tipo.

- Una vez creado el árbol binario, con información sobre todos los campos del registro, éste puede evaluarse mediante la función "**evaluar_offsets**" calculando el desplazamiento (*offset*) que corresponde a cada una de los campos con respecto a la dirección base. Esta función tiene como argumento y devuelve un puntero al nodo raíz del árbol binario.

- Tener en cuenta que esta estructura es una expresión de tipo y que después este tipo se podrá asignar a variables concretas. Cada variable proporciona una dirección base que servirá como referencia a los *offsets* calculados.

Lcampos --> Lident : Tipo

Lcampos.n = crea_arbol_bin(Lident.n, Tipo.n)
Lcampos.tamaño = Lident.conta * Tipo.n->tamaño

| Lcampos ; Lident : Tipo

$Lcampos_1$.n = añade_arbol_bin($Lcampos_2$.n, Lident.n, Tipo.n)
$Lcampos_1$.tamaño= $Lcampos_2$.tamaño+Lident.conta*Tipo.n->tamaño

Lident --> ident

Lident.n= crea_lista(ident.lex)
Lident.conta= 1

| Lident , ident

$Lident_1$.n=añade_lista(ident.lex, $Lident_2$.n)
$Lident_1$.conta = $Lident_2$.conta + 1

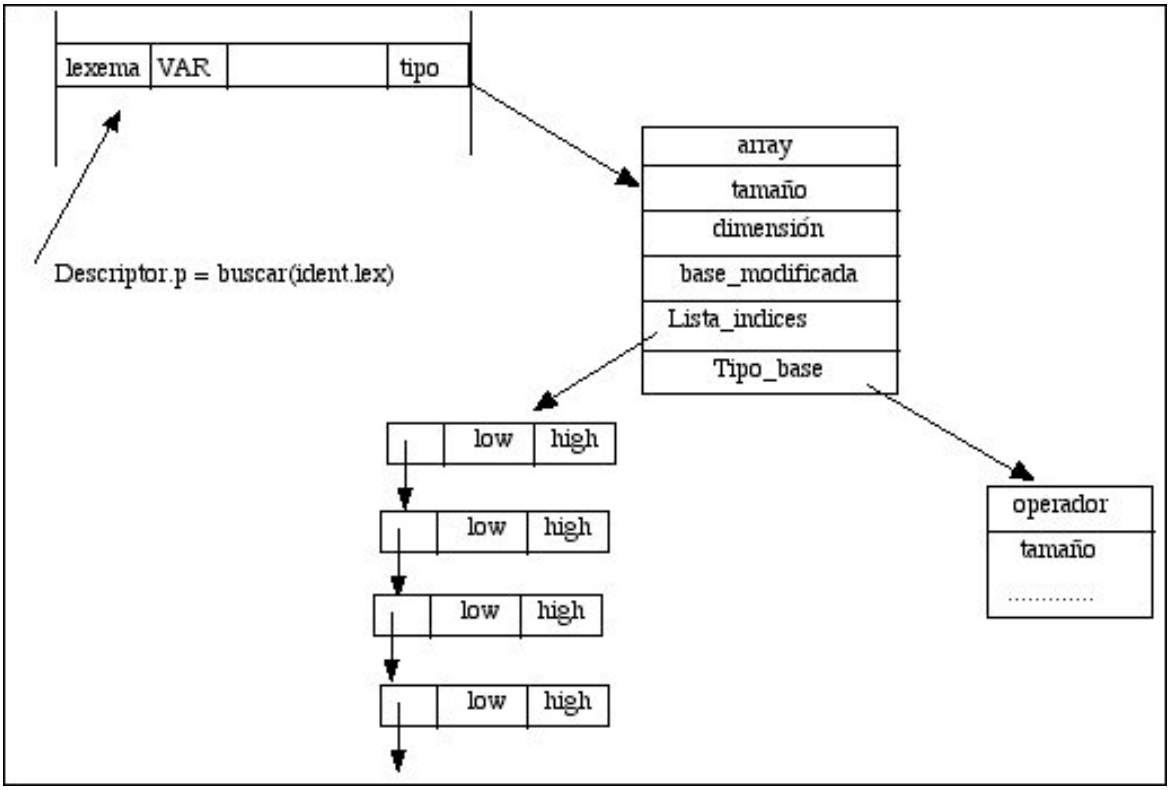
Crea una lista con los lexemas de los campos y los va contando

discusión: considerar otras estructuras, distintas del árbol binario, para representar el producto cartesiano de campos

1.4.- Ejemplo de la incorporación del tipo array (Pascal) y construcción de la expresión de tipos

Dtipos ---> TYPE ident: Tipo
Dvar ---> VAR ident: Tipo
Tipo ---> integer | real | char | boolean | ident
Tipo --> ^ Tipo
Tipo --> *record* Lcampos *end*
Lcampos --> Lident : Tipo
| Lcampos ; Lident : Tipo
Lident --> ident
| Lident , ident
Tipo --> *array* [Lindices] *of* Tipo
Lindices --> Indice
| Lindices , Indice
Indice --> numero
| numero ... numero

la categoría sintáctica **Lindices** sirve para representar el producto cartesiano de los conjuntos de índices



Tipo --> ***array*** **[** Lindices **]** ***of*** Tipo

$Tipo_1$.n = crea_array_nodo(*array*, Lindices.n, $Tipo_2$.n)
$Tipo_1$.n->tamaño = Lindices.tamaño * $Tipo_2$.n->tamaño
$Tipo_1$.n->dimension = Lindices.dimension

- la información sobre el tamaño del *array* (la cantidad de memoria necesaria para almacenar el *array*) y sobre las dimensiones del *array* podemos guardarla en el nodo raíz de la estructura creada (o evaluar la expresión de tipos cada vez que sea necesario).

$Tipo_1$.n->basemodificada = Lindices.v * $Tipo_2$.n->tamaño

- También podemos guardar la dirección base modificada del *array* si en nuestro lenguaje se puede utilizar un valor distinto de cero para los límites inferiores del *array*.
  De esta forma facilitamos la generación de código.

- **Lindices.v** es un atributo sintetizado cuyo valor depende del valor de los límites inferiores declarados en cada una de las

dimensiones del *array*. La forma en que se obtiene este atributo se justifica más adelante.

- Cuando declaremos una variable de este tipo tendremos que utilizar la dirección base de la variable y la "base_modificada" del tipo para obtener la parte "estática" de la fórmula de acceso a los elementos de un *array* (con límites inferiores distintos de cero).

Lindices --> Indice

Lindices.n = crea_nodo(null, Indice.low, Indice.high)
Lindices.tamaño = Indice.tamaño
Lindices.dimension = 1
Lindices.v = Indice.low (l_1 ; fórmulas de acceso al array)

- El atributo Lindices.n apunta a la cabeza de la lista que representa el producto cartesiano de los índices del *array*

- El atributo **Lindices.dimension** sintetiza la dimensión del array y sirve para facilitar la comprobación de tipos.

- Para representar el producto cartesiano de los índices del array utilizamos una lista enlazada.

| Lindices , Indice

$Lindices_1$.n = añadir_nodo($Lindices_2$.n, Indice.low, Indice.high)
$Lindices_1$.tamaño = $Lindices_2$.tamaño * Indice.tamaño
$Lindices_1$.dimension = $Lindices_2$.dimension + 1
$Lindices_1$.v = $Lindices_2$.v * Indice.tamaño + Indice.low
(<anterior> * nk + lk ; para el acceso al array)

añadir_nodo : el argumento **$Lindices_2$.n** es la cabeza de la lista, recorre la lista hasta el final y coloca el nuevo nodo al final de la lista de dimensiones de forma que el orden de los índices se mantiene.
$Lindices_1$.n = **$Lindices_2$.n** ambos punteros apuntan a la cabeza de la lista creada

Indice --> numero
Indice.low = 1 **/*** en C I.low = 0 */
Indice.high = numero.val /* en C I.high= numero.val-1 */
Indice.tamaño = numero.val /* nk */

| numero ... numero

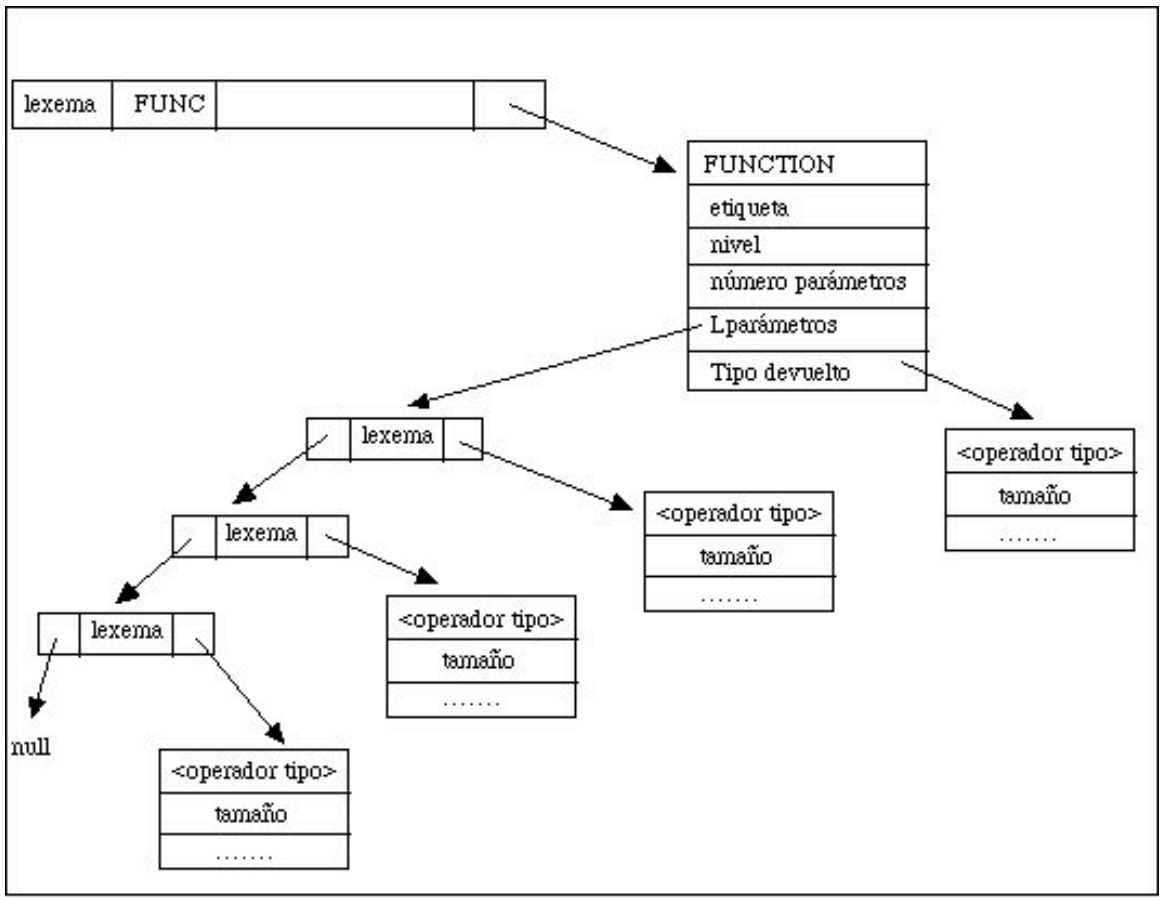
Indice.low = numero$_1$.val
Indice.hight = numero$_2$.val
Indice.tamaño = numero$_2$.val - numero$_1$.val + 1 /*nk */

ejercicio:

- construir expresiones de tipos array como ejemplo tomándolos de un programa Pascal
- desarrollar el tipo array siguiendo el modelo de C

1.5.- Ejemplo de la incorporación del tipo función (Pascal) y construcción de la expresión de tipos

Siguiendo el modelo de Pascal, la categoría sintáctica que introduce las funciones no está al mismo nivel que la categoría sintáctica que introduce los tipos construidos.



Func --> ***function*** ident (Lparametros) : Tipo

Func.p = añadir_TS(ident.lex);
if Func.p<>null
then
Func.p->clase=FUNC
Func.p->tipo=
crea_function_nodo(Lparametros.n, Tipo.n, Lparametros.numerop)
Func.p->etqcódigo = *P.etiq*
Func.p->nivel = *P.nivel_actual* *(*atención no coincide con el esquema*)*
else
error

- Añadir el nombre de la función a la tabla de símbolos puede fracasar dependiendo de las reglas que en nuestro lenguaje regulen el uso de nombres de funciones.
- **Lparametros.numerop**, al nodo del tipo función le añadimos un campo con el número de parámetros para facilitar la comprobación de tipos y generación de código

- Nuestro sistema de comprobación de tipos puede limitar los tipos que pueden ser parámetros de una función o que pueden ser devueltos por la función.
- Los atributos remotos "etiqueta" y "nivel_actual" sirven para generar la instrucción de llamada a la función y la construcción de su marco, o registro de activación, en memoria.
  Normalmente se guardarán en la propia tabla de símbolos.
- Está implícita la idea de que el significado de una función es una secuencia de instrucciones y el entorno (variables y sus ligaduras) en el que estas instrucciones se ejecutan.
- Para representar el producto cartesiano de los parámetros de la función utilizamos una lista enlazada.

Lparametros --> /*cadena vacía*/

Lparametros.n =null
Lparametros.numerop=0

Lparametros --> Lparametros ; Lident : Tipo

$Lparametros_1$.n = añade_a_listap($Lparametros_2$.n, Lident.n, Tipo.n)
$Lparametros_1$.numerop = $Lparametros_2$.numerop + Lident.conta

la función "añade_a_listap" añade parámetros al final de la lista de forma que el primer elemento de la lista es el primer parámetro.

Si la lista está vacía crea la cabeza de la lista y devuelve un puntero a la misma **$Lparametros_1$.n,** en otro caso añade los nuevos parámetros al final el nodo creado. **$Lparametros_1$.n** = **$Lparameros_2$.n**, ambos punteros apuntan a la cabeza de la lista.

Lident --> ident /* igual que en el tipo record*/

Lident.n = crea.lista(ident.lex)
Lident.conta= 1

Lident , ident

$Lident_1$.n = añade.lista(ident.lex, $Lident_2$.n)
$Lident_1$.conta = $Lident_2$.conta + 1

2.- Comprobador de tipos

El comprobador de tipos es similar al que teníamos en nuestro lenguaje mínimo, pero ahora incluye tipos construidos y nombrados.

El comprobador de tipos utiliza las expresiones de tipos previamente construidas (al procesar las declaraciones).

Las funciones siguen siendo un caso especial. Una especificación completa dependería del papel que quisiésemos dar a las funciones en nuestro lenguaje (hasta que punto nuestro lenguaje imperativo incorpora características de un lenguaje funcional).

2.1 Ampliación de la gramática.

Introducimos la categoría sintáctica *Descriptor* para "describir" sintácticamente el acceso a los componentes de un tipo construido.

Ampliamos la gramática :

I --> Descriptor := Exp
.....
Exp --> ....
.....
Fact --> <*Literal*>
Fact --> Descriptor
Fact --> ident(Lparametros)

Descriptor --> ident
Descriptor --> Descriptor**^**
Descriptor --> Descriptor**.**ident
Descriptor --> Descriptor**[**Lexp**]**

Lexp --> Exp
Lexp --> Lexp, Exp

Lparametros --> /* vacía */
Lparametros --> Lparametros, Exp

2.2 Gramática de atributos

El comprobador de tipos necesita acceder al tipo de cada elemento del lenguaje y verificar que su uso está de acuerdo con el contexto.

Los atributos **X.tipo** nos dan acceso a la representación de las "expresiones de tipo" independientemente de como se implemente la equivalencia de tipos (son punteros al nodo X del árbol que describe la expresión de tipos).

Inst --> Descriptor := Exp

if *equivalente*(Descriptor.tipo, Exp.tipo)
then Inst.tipo = ok
else Inst.tipo = error

.....
Exp --> ....
.....

Factor --> Descriptor

Factor.tipo = Descriptor.tipo

Factor --> Num | True | False | ....

Factor.tipo = <*Literal.tipo*>

Descriptor --> ident

Descriptor.p = puntero al lexema encontrado al buscar en la tabla de símbolos
Descriptor.tipo = puntero al tipo del descriptor

Descriptor.p = *buscar*(ident.lex)
if (Descriptor.p<> null) *and*
(Descriptor.p->clase==VAR)
then Descriptor.tipo = Descriptor.p->tipo
else Descriptor.tipo = error

/* El identificador debe ser el nombre de una variable cuyo tipo es accesible desde la tabla de símbolos.
Podemos ampliar esta especificación para que nuestro lenguaje admita constantes definidas en la tabla de símbolos */

2.3 Punteros

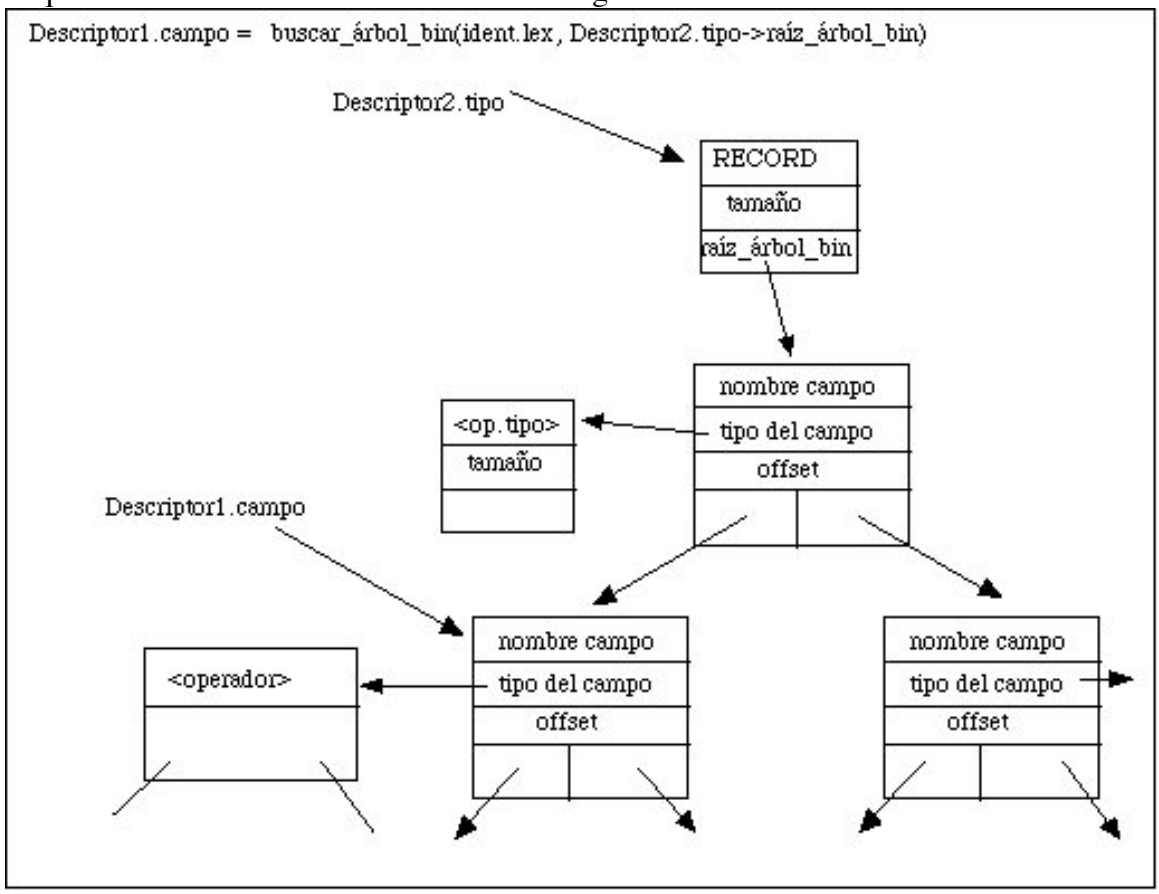
Descriptor --> Descriptor ^

if (Descriptor$_2$.tipo<> error) ***and***
Descriptor$_2$.tipo->op == *POINTER*)

then Descriptor$_1$.tipo = Descriptor$_2$.tipo->tipo_base
else Descriptor$_1$.tipo = error

2.4 Registros

Esquema del acceso a la información en un registro



Descriptor --> Descriptor **.** ident

Descriptor.tipo = puntero al tipo del descriptor
Descriptor.campo = puntero al nodo campo encontrado por la función que busca en el árbol binario

$Descriptor_1$.campo=
buscar_árbol_bin(**ident.lex, $Descriptor_2$.tipo->raíz_árbol_bin**)

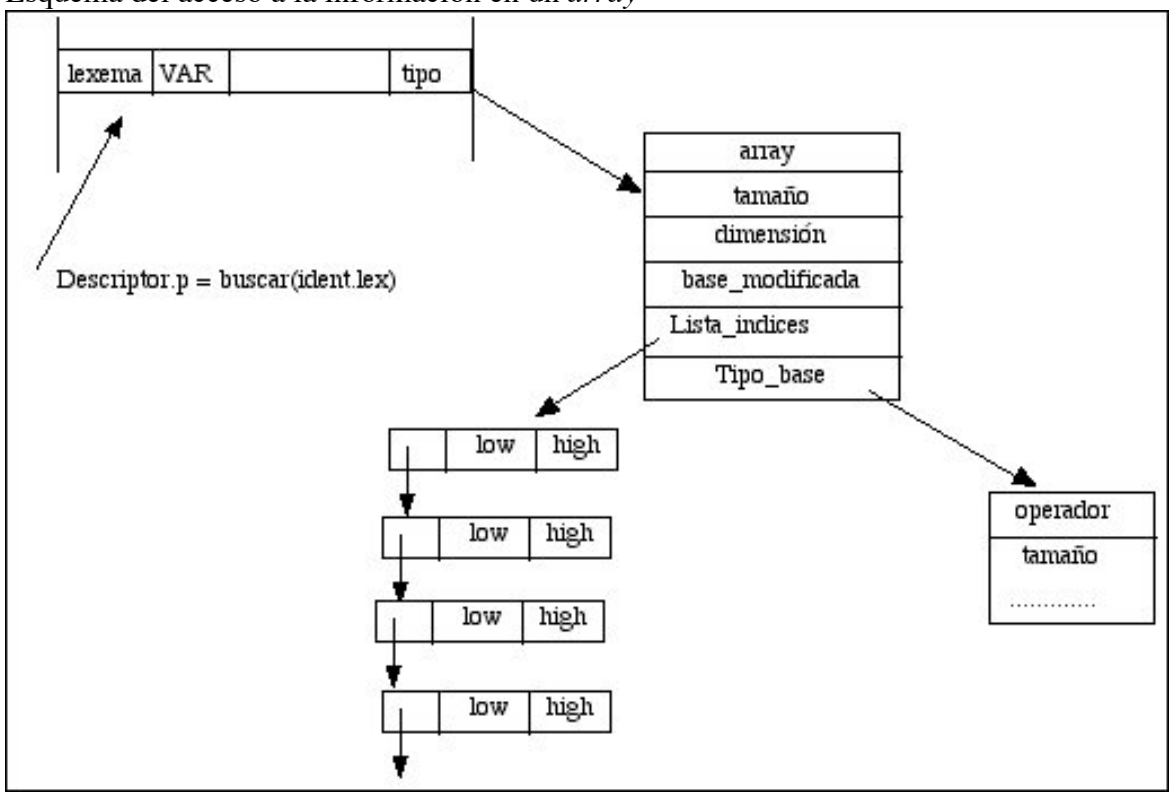
if ($Descriptor_2$.tipo<> error) ***and***
($Descriptor_2$.tipo->operador == *RECORD*) ***and***
($Descriptor_1$.campo<> null)

then $Descriptor_1$.tipo = $Descriptor_1$.campo->tipo
else $Descriptor_1$.tipo = error

la función "buscar_árbol_bin" busca el lexema que da nombre a un campo en el árbol binario. Devuelve el puntero al nodo encontrado ó null.

2.5 Arrays

Esquema del acceso a la información en un *array*



Descriptor --> Descriptor[L.exp]

Descriptor.tipo = puntero al tipo del descriptor
Lexp.tipo = tipo de la lista de expresiones (debe ser escalar o error)
Lexp.dim = dimensión del array

if **(Desriptor$_2$.tipo<>error)** ***and***
(Descriptor$_2$.tipo->op == *ARRAY*) ***and***
(Lexp.tipo == escalar) ***and***
(Lexp.dim == Descriptor$_2$.tipo->dimensión)

then **Descriptor$_1$.tipo = Descriptor$_2$.tipo->tipo_base**
else **Descriptor$_1$.tipo = error**

no comprobamos que las expresiones calculadas están entre los límites de cada una de las dimensiones, esa comprobación de tipo debe hacerse dinámicamente (veremos un ejemplo en la generación de código).

L.exp --> Exp

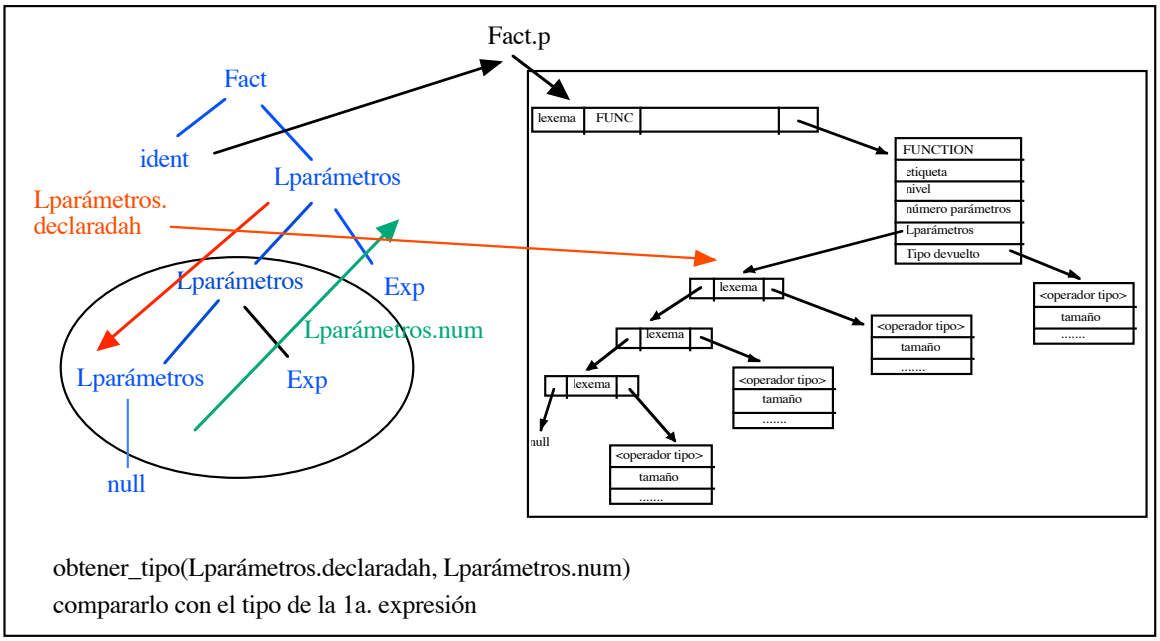
Lexp.dim = 1
if ***escalar*****(Exp.tipo)**
then Lexp.tipo = escalar
else Lexp.tipo=error

| Lista.exp, Exp

Lexp$_1$.dim=Lexp$_2$.dim +1
if (Lexp$_2$.tipo==escalar) ***and*** ***escalar*****(Exp.tipo)**
then Lexp$_1$.tipo = escalar
else Lexp$_1$.tipo = error

2.6 Funciones

Esquema del acceso a la información de una función



Contexto de uso: la llamada a una función es un factor dentro de una expresión.

Fact --> ident(Lparametros)

Fact.p = puntero al lexema (**ident**) encontrado en la tabla de símbolos
Fact.tipo = el tipo del factor
Lparametros.tipo = atributo sintetizado. Es **ok** si todos los tipos de las "expresiones" de llamada coinciden ordenadamente con los tipos declarados.
Lparametros.num = atributo sintetizado. Contador de parámetros en la llamada a la función, empieza en cero y termina con el número total de parámetros en la llamada.
Lparametros.declaradah = atributo heredado. Es un atributo que sirve para conectar la declaración de la función con el uso que hacemos de la función en la expresión.
Es un puntero a la declaración de la función en la tabla de símbolos que pasamos como atributo heredado a la lista de parámetros actuales.

Fact.p = buscar(ident.lex);

Lparametros.declaradah = Fact.p->tipo->Lparametros

if (Fact.p<> null) ***and***
(Fact.p->clase==FUNC) ***and***
(Lparametros.tipo==ok) ***and***
(Fact.p->tipo->numerop==Lparametros.num)

then Fact.tipo = Fact.p->tipo->Tresultado
else Fact.tipo = error

Lparametros --> /*cadena vacía*/
Lparametros.num = 0
Lparametros.tipo = ok

/* Si el número de parámetros es cero, siempre damos por correcto el tipo de la lista de parámetros, bastará con comprobar después que el número de parámetros también es cero en la declaración */

| Lparametros , Exp

$Lparametros_1$.num = $Lparametros_2$.num + 1;

if (*$Lparametros_2$*.tipo = ok) ***and***
equivalente **(Exp.tipo,**
obtenerTipo**($Lparametros_1$.declaradah,**
$Lparametros_1$.num)
then $Lparametros_1$.tipo = ok
else $Lparametros_1$.tipo = error;

$Lparametros_2$.declaradah = $Lparametros_1$.declarada.h;

/* la función "obtenerTipo" tiene como parámetros el puntero a la lista de parámetros declarada y el lugar que ocupa en esa lista el parámetro que se quiere comprobar (en nuestro caso **$Lparametros_1$.num** tiene el número de orden que ocupa Exp en la lista de la llamada) y que tiene que ser equivalente con el tipo de la expresión */