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

ASPECTOS PRÁCTICOS

3.- Generación de código: acceso a los elementos de los tipos construidos

Cápitulos 7 y 8 del Aho

- Semántica informal de *Descriptor* y *Factor*:
  - un descriptor es, o describe, una dirección
  - un factor es un valor, es decir, el contenido de una dirección.

- El código generado para acceder a un elemento de un *record* o de un *array* puede optimizarse en una fase posterior.

- Funciones. Como en los apartados anteriores quedan detalles por precisar con respecto a las funciones.
  - El más importante es el significado de las direcciones asociadas a las variables. Si el lenguaje tiene funciones y las variables se

declaran dentro de una función, las direcciones asociadas a las variables son direcciones relativas al bloque de memoria asociado con la ejecución de la función: su "marco", o "registro de activación".

- Para poder acceder a las variables locales y no locales, una variable tiene que tener asociado el "nivel" en que ha sido declarada y, las instrucciones de la máquina-p que manipulan las direcciones, necesitarán un segundo parámetro que indique el nivel de la variable.

3.1 Primero necesitamos ampliar/modificar la gramática:

.....
I --> new(ident) *(* se puede generalizar a un descriptor*)*
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 */
Lparámetros --> Lparametros, Exp

3.1.1 Descriptor

Atributos de la categoría sintáctica Descriptor:

- **Descriptor.c** , código que al ejecutarse dejará en la cima de la pila de operandos la dirección que "describe"
- **Descriptor.p**, puntero de acceso a la tabla de símbolos
- **Descriptor.tipo**, puntero al nodo raíz de la expresión de tipos, necesario para acceder a los datos del tipo asociado al descriptor(datos necesarios para generar el código)
- **Descriptor.campo,** puntero que permite acceder a la descripción del campo de un registro.

3.1.2 Modificaciones en la Máquina-P

- Ampliamos nuestra maquina-P con:

1.- Un bloque de **memoria dinámica**.

- Para facilitar la implementación el espacio de direcciones de la memoria de datos estática y dinámica es el mismo.
- Es decir, podemos tener un único *array* de memoria, y desde un valor fijo, p.e. aproximadamente desde la mitad y creciendo hasta el final, lo reservamos para memoria dinámica.
- Otra solución sería que la memoria dinámica creciese desde el final hacia posiciones inferiores y la estática al revés.

2.-Un registro, **H**, que apunta a la 1ª posición de memoria libre en el bloque de memoria dinámica (para simplificar no se implementa un recolector de basura)

discusión: diferentes formas de concebir la máquina virtual

3. - Nuevas instrucciones de la máquina-p:

apila_ind
desapila_ind
apilaH
incrementaH ***<posiciones de memoria>***

<P, [d | S], M, i, h, r>	**<P, [M(d) | S], M, i+1, h, r>**	**I(P,i) = apila_ind**
<P, [v, d | S], M, i, h, r>	**<P, S, M[d <- v], i+1, h, r>**	**I(P,i) = desapila-ind**
<P, S, M, i, h, r>	**<P, [h | S], M, i+1, h, r>**	**I(P,i) = apilaH**
<P, S, M, i, h, r>	**<P, S, M, i+1, h+v, r>**	**I(P,i) = incrementaH v**

/* desarrollaremos primero un ejemplo, después veremos como relacionamos estas operaciones con las estructuras sintácticas */

ejercicio: representar los árboles sintácticos de las instrucciones y decorarlos con los atributos

.....
VAR

(declaramos la variable fecha como un puntero al tipo básico integer*)*

Fecha: ^integer;

(creamos la entrada en la tabla de símbolos *)*

.....
new(Fecha);

*(*asignamos a la variable fecha una dirección, sin usar, de la memoria dinámica*)*

I. p:=busca(ident.lex)

apila I.p->dir
apilaH
desapila_ind
incrementaH I.p->tipo->tipo_base->tamaño

*(*La variable "Fecha" contiene la dirección de la memoria dinámica puesta en la pila por apilaH *)*

.....
Fecha^:= 15;

*D. p:= busca(ident.lex) (*busca "Fecha" en la TS*)*

*apila D.p->dir (*generada por la categoría D -->ident*)*

```
        apila_ind       (*generada por la categoría D ->D^ *)
                        (*el resultado es poner en la cima de la pila la
                        dirección de "Fecha" en la memoria dinámica,
                        lugar donde vamos a guardar el valor de la
                        expresión del lado derecho de la asignación*)
        apila 15  (* Exp.c *)
        desapila_ind
```

.....
dia := Fecha^ + 2;

```
                D.p:= busca(ident.lex) (*busca "dia" en TS*)

        apila D.p->dir      (*generada  categoría D ->ident(dia)*)
                            (*generación del código de la expresión
                            en el lado derecho*)
                I. D:= busca(ident.lex)  (*busca "Fecha" en TS*)

        apila  D.p->dir   (* dirección estática de "Fecha" *)
        apila_ind         (*obtengo la dirección dinámica *)
        apila_ind         (*obtengo el contenido *)
        apila   2
        suma
        desapila_ind      (* desapila en la dirección de "dia"*)
```

ejercicio: representar los árboles sintácticos de las instrucciones y decorarlos con los atributos

/* fin del ejemplo */

3.2 Punteros, Descriptor, Factor

I --> new(ident)

I.p = busca(ident.lex)
(*suponemos que el comprobador de tipos ha verificado que ident ha sido declarado y que es del tipo *pointer**)

I.c = apila I.p->dir || apilaH || desapila_ind
|| incrementaH I.p->tipo->tipo_base->tamaño

la variable identificada por **ident.lex** contiene una dirección de la memoria dinámica

discusión: sintaxis alternativas para crear variables dinámicas

Descriptor --> ident

Descriptor.p = busca(ident.lex)
Descriptor.c = apila Descriptor.p->dir

Obtenido por el comprobador de tipos

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

Factor --> Descriptor

Factor.c = Descriptor.c || apila_ind

El atributo **Factor.c** es el código para obtener el valor de la dirección señalada por el descriptor (o directamente en el caso de las constantes).

Con respecto a los factores literales no hay variación en la generación de código.

Factor --> Num

Factor.c = apila Num.val

Descriptor --> Descriptor^

$Descriptor_1$.c = $Descriptor_2$.c || apila–ind

Comprobador de tipos

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

I --> Descriptor := Exp

I.c = Descriptor.c || E.c || desapila_ind

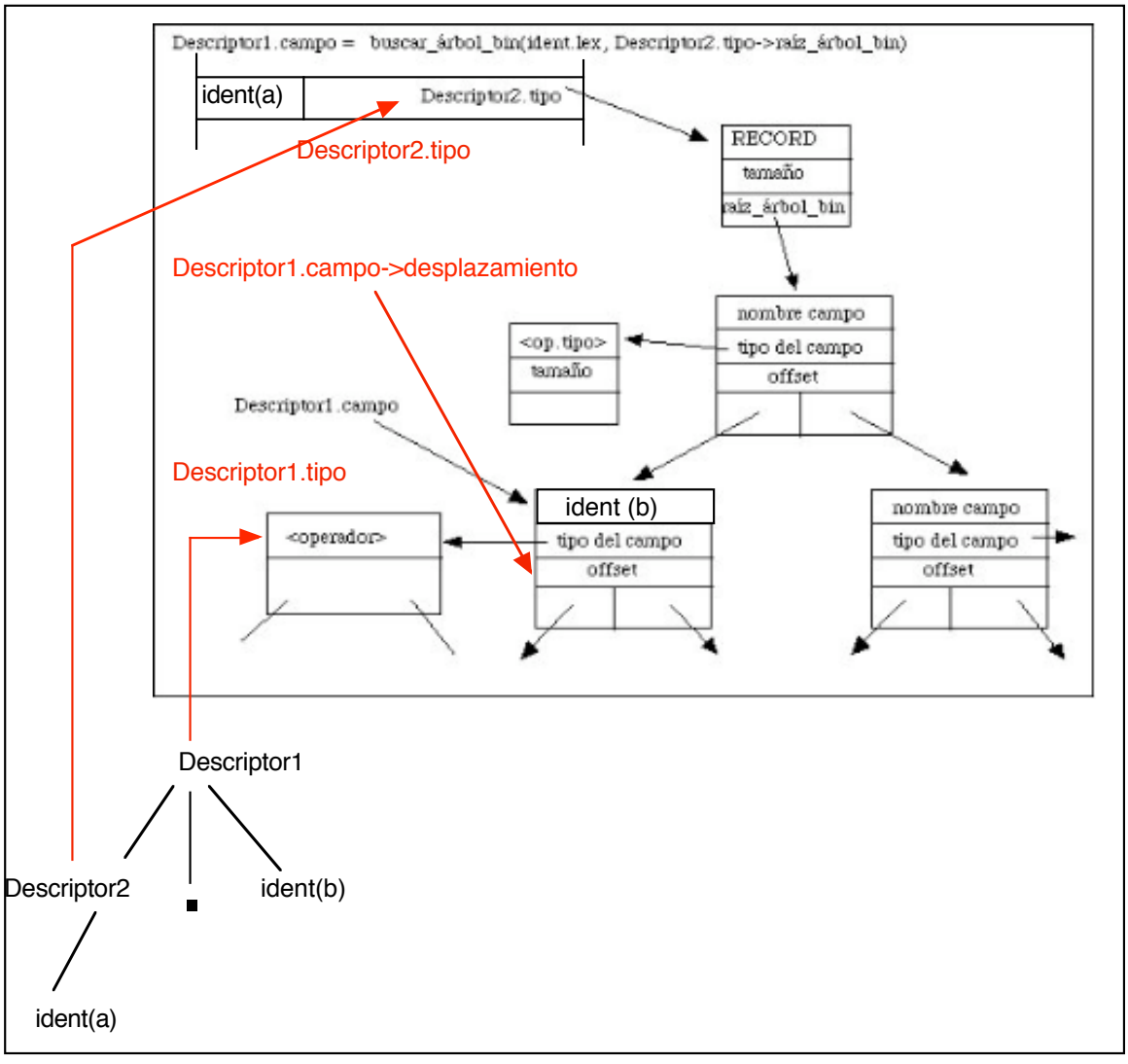
Atributo **Descriptor.c** es el código que al ejecutarse generará en la cima de la pila de operandos la dirección donde se desapila, o almacena, el resultado de evaluar la expresión.

3.3 Registros

Con esta gramática vamos a tratar el acceso a los campos de un *record* dinámicamente.

El compilador siempre podrá, en un paso posterior, optimizar este código de acceso ejecutando las instrucciones aritméticas que solo involucren operandos conocidos en tiempo de compilación.

Esquema del acceso a los campos de un registro



Descriptor --> Descriptor **.** ident

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

$Descriptor_1$.c = $Descriptor_2$.c ||
apila $Descriptor_1$.campo->*offset* || suma

Obtenido por el comprobador de tipos

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

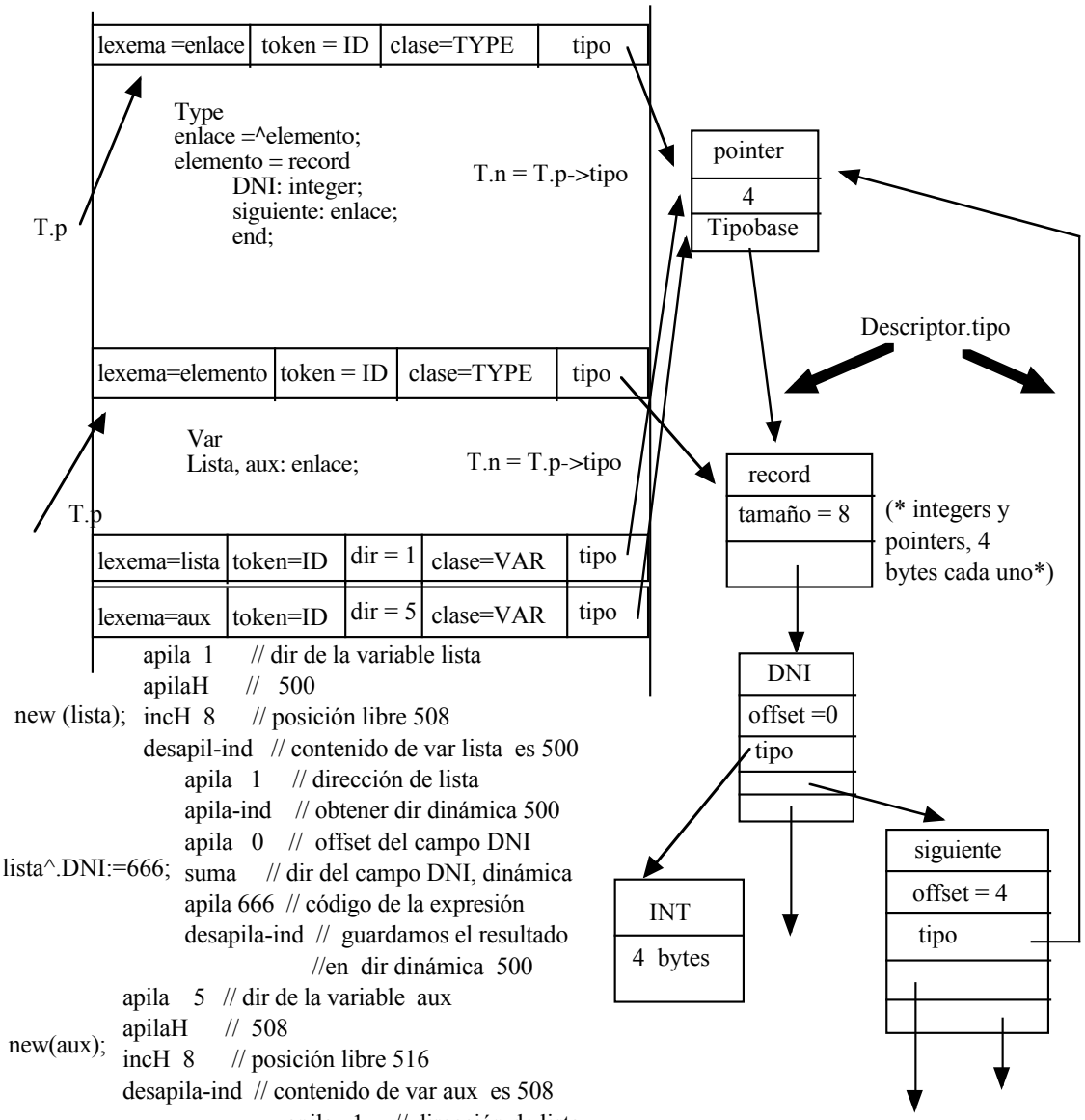
El atributo $Descriptor_1$.campo apunta al nodo del campo y nos permite acceder al desplazamiento y al tipo del campo.

En el atributo $Descriptor_2$.c tenemos una dirección base a la que sumamos un desplazamiento, *offset*, para llegar a la dirección objetivo.

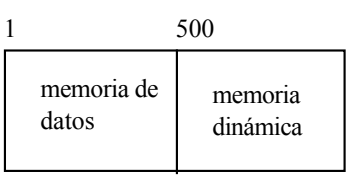
Ejemplo

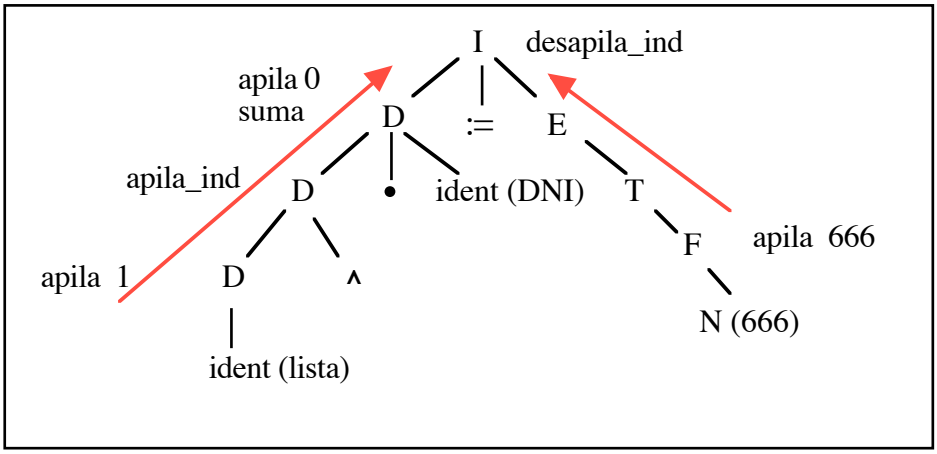
```
TYPE
        enlace = ^elemento;
        elemento = record
                        DNI: integer;
                        siguiente: enlace;
                   end;
......
VAR
        lista, aux: enlace;
......
new(lista);
lista^.DNI :=666;
new(aux);
.............
lista^.siguiente :=aux;
```

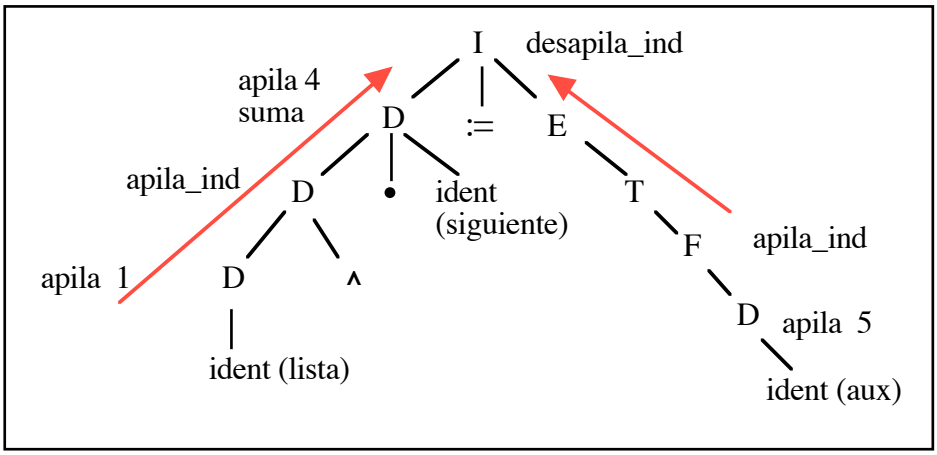
.....
lista^.siguiente^.DNI:= 32567; (* equivalente a aux^.DNI:=32567 *)
.....

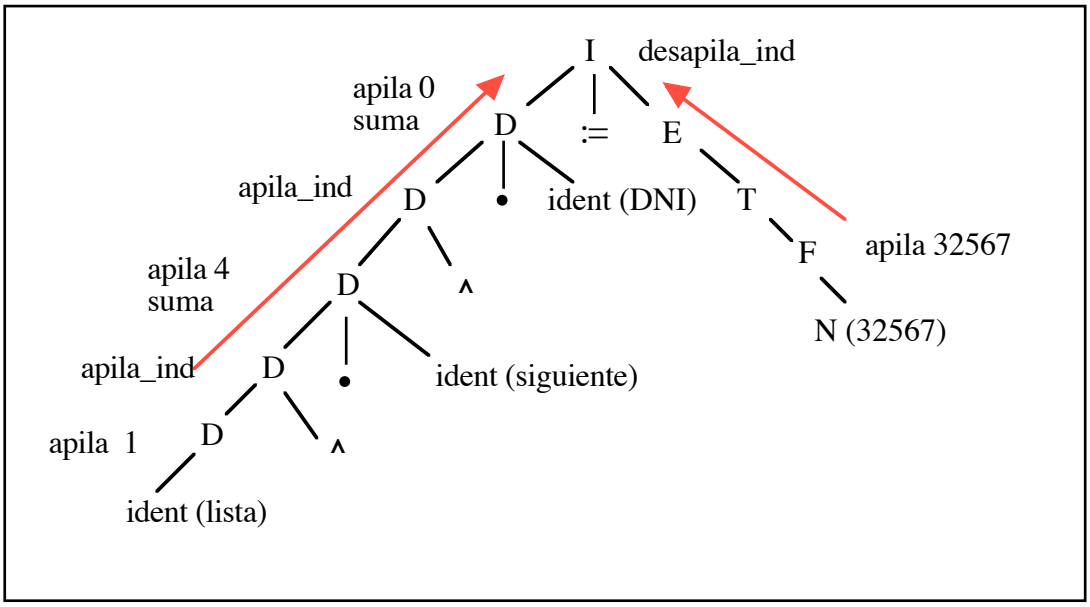



```
                apila  1     // dir de la variable lista
                apilaH      //   500
new (lista);    incH  8     // posición libre 508
                desapil-ind   // contenido de var lista  es 500
                    apila   1     // dirección de lista
                    apila-ind     // obtener dir dinámica 500
                    apila   0     //  offset del campo DNI
lista^.DNI:=666;    suma      // dir del campo DNI, dinámica
                    apila 666  // código de la expresión
                    desapila-ind  //  guardamos el resultado
                                  //en  dir dinámica  500
                apila    5   // dir de la variable  aux
new(aux);       apilaH      //  508
                incH  8      // posición libre 516
                desapila-ind  // contenido de var aux  es 508
                                apila   1     // dirección de lista
                                apila-ind     // obtener dir dinámica ,500
                                apila   4     //  offset del campo siguiente
lista^.siguiente:=aux;          suma      // obtener dir del campo siguiente, dinámica
                                apila  5  // código de la expresión, dir de aux
                                apila-ind // dir dinámica, 508
                                desapila-ind  //  campo siguiente en lista apunta a aux, 508
                                    apila   1     // dirección de lista
lista^.siguiente^.DNI:=32567;       apila-ind     // obtener dir dinámica, 500
                                    apila   4     //  offset del campo siguiente
                                    suma      // obtener dir del campo siguiente en lista: 504
                                    apila-ind   // obtener contenido del campo siguiente, 508 (donde apunta)
                                    apila  0  // offset del campo  DNI
                                    suma  // dirección del campo DNI en  el segundo elemento: 508
                                    apila  32567
                                    desapila-ind  //  valor del DNI en el segundo elemento de la lista
```





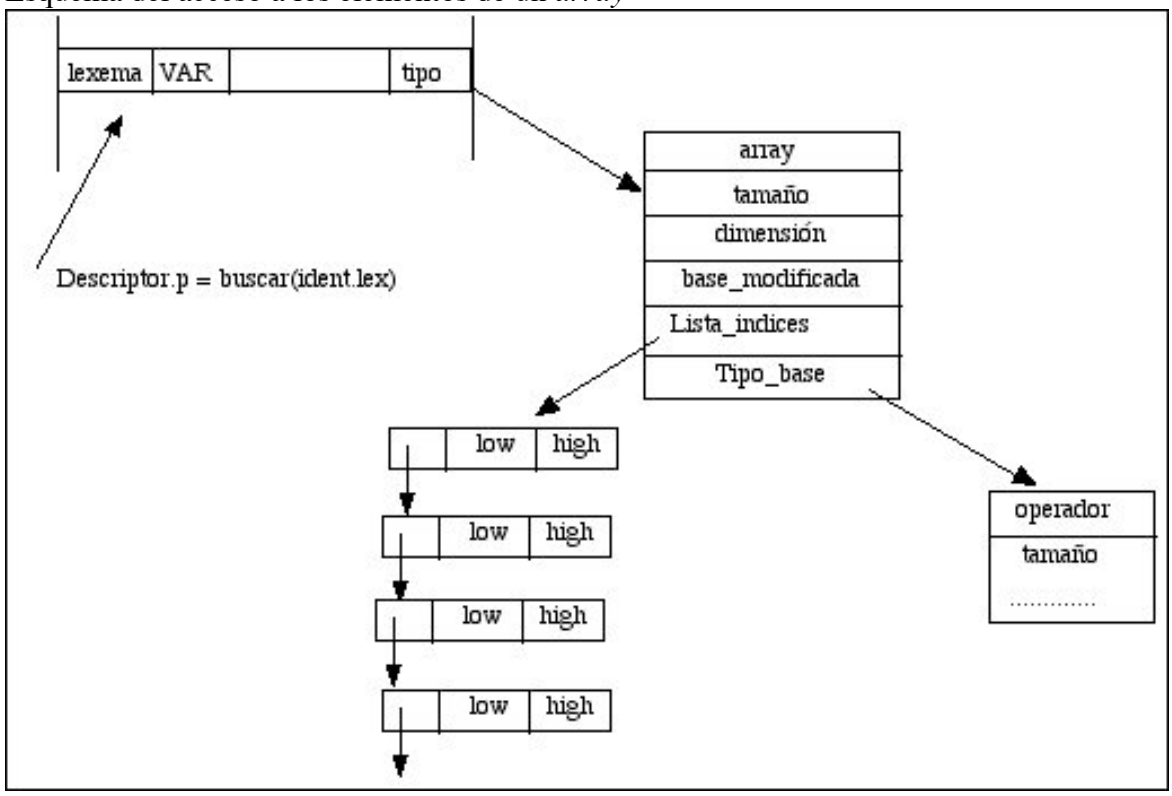




/* fin del ejemplo */

3.4 Arrays

Esquema del acceso a los elementos de un *array*



Descriptor --> Descriptor[Lexp]

Lexp.indicesh = Descriptor$_2$.tipo->Lista_indices

Atributo heredado para acceder a la información sobre los índices del *array*.

Descriptor$_1$.c =

Descriptor$_2$.c (* dir.base del array*)
|| apila Descriptor$_2$.tipo->base_modificada
|| Resta

(* [dir.base **-** [(… (l_1 * n_2 + l_2)*n_3 +l_3) …) *n_k + l_k)] *Tipobase->tamaño]
+ [(… (i_1*n_2 +i_2)*n_3+i_3)* …)* n_k + i_k)* Tipobase->tamaño] *)

|| Lexp.c
|| apila Descriptor$_2$.tipo->tipo_base->tamaño
|| Multiplica
|| Suma

Nota aclaratoria:
para acceder a una posición concreta del *array* hay que sumar a la dirección base del *array* una cantidad función de los índices actuales

(calculada dinámicamente en Lexp.c $[(...(i_1*n_2 +i_2)*n_3+i_3)*...)* n_k + i_k)]$ y multiplicada por el tamaño del tipo base).

Pero si los límites iniciales de cada una de las dimensiones del array son distintos de 0 hay que restar una cantidad a la dirección base que es función de estos límites iniciales y del tamaño del tipo base del *array*. Esta cantidad es la que llamamos "base modificada"

$Descriptor_2$.tipo->base_modificada =
$[(\ldots(l_1 * n_2 + l_2)*n_3 +l_3)*n4) *n_k + l_k)]$ * Tipo_base->tamaño

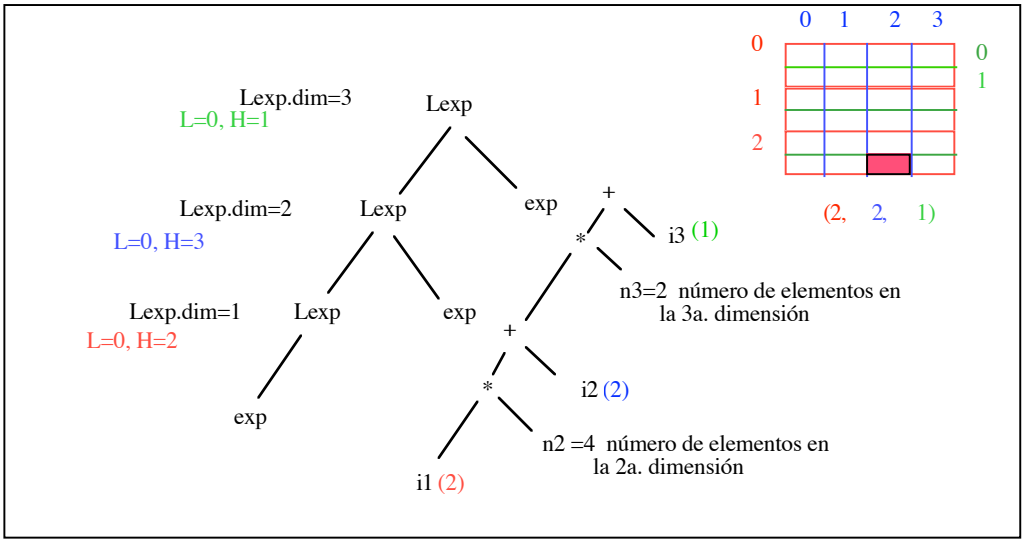
---

L.exp → Exp

Lexp.dim = 1 (* localizador de la dimensión en la declaración *)
Lexp.H = obtenerH(Lexp.indicesh, Lexp.dim)
Lexp.L = obtenerL(Lexp.indicesh, Lexp.dim)
Lexp.c = Exp.c (* i_1 *)
|| copia || apila Lexp.H || menorigual || ir-falso *<error>*
|| copia || apila Lexp.L || mayorigual || ir-falso *<error>*
(* comprobación de tipos dinámica*)

| L.exp , Exp

$Lexp_2$.indicesh = $Lexp_1$.indicesh
(* puntero a la declaración de índices *)
$Lexp_1$.dim = $Lexp_2$.dim + 1 (* localizador de la dimensión *)
$Lexp_1$.nk = obtener_nk($Lexp_1$.indicesh, $Lexp_1$.dim)
(* n_k = k_High - k_Low + 1 *)
(* se puede asignar a $Lexp_1$ ó $Lexp_2$ *)
$Lexp_1$.H = obtenerH($Lexp_1$.indicesh, $Lexp_1$.dim)
$Lexp_1$.L = obtenerL($Lexp_1$.indicesh, $Lexp_1$.dim)
$Lexp_1$.c = $Lexp_2$.c || apila $Lexp_1$.nk || Multiplica || Exp.c
|| copia || apila $Lexp_1$.H || menorigual || ir-falso *<error>*
|| copia || apila $Lexp_1$.L || mayorigual || ir-falso *<error>*
(* comprobación de tipos dinámica sobre el índice Exp.c*)
|| Suma
<código anterior> *n_k + i_k



3.5 Funciones

Fact --> ident(Lparametros)

Fact.p=busca(ident.lex);
Fact.c = Lparámetros.c ||
***call f*(Fact.p->nivel) Fact.p->etqcodigo**

Nota: la instrucción de la máquinaP que llama a una función es nueva y necesariamente compleja.

- Tiene que actuar sobre el contador de programa pero también tiene que organizar en la memoria de datos el entorno en que queremos que se ejecute el código de la función llamada.
- La complejidad de esta instrucción depende de la interpretación que hagamos del paradigma funcional en nuestro lenguaje imperativo.
- Además necesitaremos otras instrucciones para la creación del área de datos (marco ó registro de activación de la función llamada).
- También necesitaremos definir como se producirá el "retorno" del valor calculado por la función

Todos estos problemas los trataremos con detalle en el tema siguiente al tratar del "paquete de apoyo" a la ejecución de funciones.

(* gestionar el resultado de la función
gestionar el número de parámetros *)

Lparametros --> /*cadena vacía*/

Lparametros.c = /* vacio */

| Lparametros , Exp
$Lparametros_1.c = Lparametros_2.c$ || E.c
|| <paso por valor ó referencia …>

no esta completamente determinado el código que tengo que generar al no estar definida la estructura de los registros de activación de las funciones. En particular si la pila de operandos está incorporada o no al registro de activación. La implementación más simple y eficiente es que esté incorporada (no habría que generar código para el paso de parámetros).

==========

Notas complementarias

1. Fórmulas para obtener el código de acceso a los elementos de un *array* (similar a los *arrays* de Pascal).

2. Ejemplo de código y comprobación dinámica de tipos en un *array*

----

1. Fórmulas para obtener el código de acceso a los elementos de un array.

- Al permitir expresiones en los índices de un *array*, estas variables son en realidad dinámicas, en el sentido de que tendremos que determinar durante la ejecución, al evaluar las expresiones, cual es la posición de la memoria a la que accedemos.

- Esto también implica comprobación dinámica de tipos si queremos garantizar que los índices calculados se encuentran dentro de los límites declarados del *array*.

- El lenguaje C da un paso más y considera que los *arrays* son "de facto" un tipo de variables dinámicas, punteros, de los que se diferencian por la forma en que se reserva memoria para ellos y por la zona de memoria en la que se almacenan.

Declaración de los límites de un *array* unidimensional: (low, high)

explícitamente: ident[low … high] ➔ tamaño= high - low +1

implícitamente: ident[h] equivale a ident[1 ... h] ➔ tamaño = h

ident[h] equivale a ident[0 … h-1] ➔ tamaño = h

Aritmética de los *arrays* siguiendo el modelo de Pascal.

¡Importante! si los límites inferiores (low) de las dimensiones del *array* se toman igual a cero los cálculos se simplifican mucho (es la opción de C)

1.1 Caso unidimensional, límites del *array* (l_1, h_1)

para acceder al elemento i_1-ésimo:

dir.base + (i_1 - l_1) * Tipobase->tamaño

reagrupando términos

[dir.base - (l_1 * Tipobase->tamaño)] + [i_1 * Tipobase->tamaño]

- El primer sumando se puede calcular evaluando la expresión de tipos
- El segundo sumando depende del valor actual del índice y hay que generar código para calcularlo dinámicamente

1.2 Caso de dos dimensiones, límites del array (l_1, h_1), (l_2, h_2)

para acceder al elemento (i_1, i_2):

dir.base + [(i_1-l_1) * n_2 + (i_2 - l_2)] * Tipobase->tamaño

donde
n_2 = h_2 - l_2 +1 = número de elementos de una fila (2ª dimensión) del array.

reagrupando

dir.base + (i_1 * n_2 - l_1 * n_2 + i_2 - l_2) * Tipobase->tamaño =
= [dir.base - (l_1*n_2 + l_2) * Tipobase->tamaño]

$+ [(i_1 * n_2 + i_2) *$ Tipobase->tamaño]

el primer sumando estático, el segundo dinámico

1.3 Caso de tres dimensiones, (l_1, h_1), (l_2, h_2), (l_3, h_3)

para acceder al elemento (i_1, i_2, i_3)

dir.base + $[(i_1 - l_1) * n_2 * n_3 + (i_2 - l_2) * n_3 + (i_3 - l_3)]$ * Tipobase->tamaño

reagrupando

dir.base+ $[i_1 * n_2 * n_3 - l_1 * n_2 * n_3 + i_2 * n_3 - l_2 * n_3 + i_3 - l_3]$ * Tipobase->tamaño =
[dir.base - $((l_1 * n_2 + l_2) * n_3 + l_3)$ * Tipobase->tamaño]
+ $[((i_1 * n_2 + i_2) * n_3 + i_3) *$ Tipobase->tamaño]

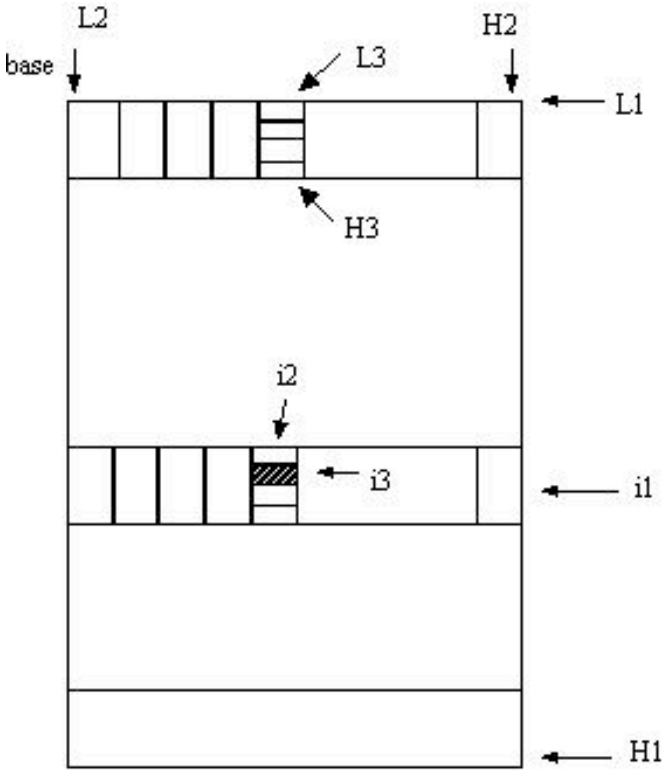
el primer sumando estático, el segundo dinámico

1.4 Caso de k dimensiones $(l_1, h_1) \ldots (l_k, h_k)$

para acceder al elemento $(i_1, i_2, \ldots, i_k)$

= [dir.base - $[(\ldots (l_1 * n_2 + l_2) * n_3 + l_3) \ldots) * n_k + l_k)$] *Tipobase->tamaño]
+ $[(\ldots (i_1 * n_2 + i_2) * n_3 + i_3) * \ldots) * n_k + i_k) *$ Tipobase->tamaño]

el primer sumando estático, el segundo dinámico



1.- caso unidimensional, límites del array (L1 , H1)
para acceder al elemento i1-ésimo:
dir. base + (i1 - L1) * Tipobase->tamaño
reagrupando
[dir. base - (L1 * Tipobase->tamaño)] + [i1 * Tipobase->tamaño]
- el primer sumando estático, el segundo dinámico

2.- caso de dos dimensiones, límites del array (L1 , H1), (L2 , H2)
para acceder al elemento (i1 , i2):
dir. base + [(i1-L1) * n2 + (i2 - L2)] * Tipobase->tamaño
n2 = H2 - L2 +1 = número de elementos de una fila (2a. dimensión) del array.
reagrupando
dir. base + (i1 * n2 - L1 * n2 + i2 - L2) * Tipobase->tamaño =
[dir. base - (L1*n2 + L2) * Tipobase->tamaño]
+ [(i1*n2 +i2)* Tipobase->tamaño]
- el primer sumando estático, el segundo dinámico

3.- caso de tres dimensiones, (L1 , H1), (L2 , H2), (L3 , H3)
para acceder al elemento (i1 , i2 , i3)
dir. base + [(i1 - L1) *n2 *n3 + (i2 - L2) *n3 + (i3 -L3)] * Tipobase->tamaño
reagrupando
dir. base+ [i1*n2*n3 - L1*n2*n3 + i2*n3 - L2*n3 +i3 -L3] * Tipobase->tamaño =
[dir. base - ((l1 * n2 + l2)*n3 +l3) * Tipobase->tamaño]
+ [((i1*n2 + i2)*n3 + i3)* Tipobase->tamaño]
- el primer sumando estático, el segundo dinámico

4. caso de k dimensiones (L1 , H1) … (Lk, Hk)
para acceder al elemento (i1 , i2, … , ik)
[dir. base - [(… (L1 * n2 + L2)*n3 +L3) …) *nk + Lk)] *Tipobase->tamaño]
+ [(… (i1*n2 +i2)*n3+i3)* …)* nk + ik)* Tipobase->tamaño]
- el primer sumando estático, el segundo dinámico

2. Ejemplo de comprobación de tipos y de generación de código.

```
…
TYPE
fecha= array [2000..2004, 1..6, 15..31] of integer;
hito=record
        codigo:integer;
        entrega: fecha;          entrega.offset=4
        end
.......
VAR
año_inicio,  dias_margen: integer;
galateaH: hito;
                                 año_inicio .dir =1
                                 dias_margen.dir =5
                                 galateaH.dir = 9
.......
año_inicio:=2001;
dias_margen:=13;
galateaH.entrega[año_inicio + 1,   5,   dias_margen+7]:= 100;
-------------------------------------------------------------------------

      apila  9          (* D --> ident(galateaH)   D.p->dir =9  *)
      apila  4          (* D1 --> D2.ident(entrega)
                           D2.campo= BuscaAB(D2->tipo, id.lex)
                           D2.campo->offset=4                    *)
      suma
      apila  816128     (* D.tipo->baseModificada=[[2000*6+1]*17+15]*4
                           [[ l1*n2+l2*]n3+l3]* D.tipo->Tipo_base->tamaño    *)
      resta
      apila 1           (* D->ident(año_inicio)  D.p->dir =1  *)
      apila_ind         (* F -> D *)
      apila  1
      suma              (* i1 = 2001 + 1 *)
      copia
      apila  2004
      <=
      ir_f  <error> si falso ir rutina de error
      copia
      apila 2000
      >=
      ir_f  <error>  si falso ir a rutina de error
      apila  6          (* n2 = 6 -1 +1 *)
      Multiplica        (*  i1*n2  *)
      apila 5           (* i2 *)
      copia
      apila  6
      <=
      ir_f  <error> si falso ir rutina de error
```

```
copia
apila 1
>=
ir_f  <error>  si falso ir a rutina de error
Suma            (*  i1*n2+ i2  *)
apila  17       (* n3 = 31 - 15 +1  *)
Multiplica      (* [i1*n2+ i2] * n3  *)
apila 5         (* D->ident(dias_margen)  D.p->dir=5 *)
apila_ind               (* F -> D *)
apila 7
Suma            (*i3 =dias_margen +7 *)
copia
apila  15
<=
ir_f  <error> si falso ir rutina de error
copia
apila 31
>=
ir_f  <error>  si falso ir a rutina de error
suma            (* [i1*n2+ i2] * n3 + i3 *)
apila 4         (* D.tipo->tipo_base->tamaño *)
Multiplica
Suma
        (* <dir_base - base_modificadaM> + [Lexp.c * D.tipo->tamaño] *)
apila 100
desapila-ind
```