Lección 1: Tablas de Símbolos

1. Introducción
2. Espacio de nombres
3. Organización
4. Lenguajes de bloques
5. Perspectiva
6. **`tabla.c` y `tabla.h`**
7. Ejercicios

Lecturas:

Scott, sección 3.3
Munchnick, capítulo 3
Aho, sessión 7.6
Fischer, capítulo 8
Holub, sección 6.3
Bennett, sección 5.1
Cooper, secciones 5.7 y B.4

La Tabla de Símbolos

- ¿porqué es importante?
  - ¿análisis léxico?
  - ¿análisis sintáctico?
  - ¿análisis semántico?
  - ¿generación de código?
  - ¿optimización?
  - ¿ejecución?

```
int a, 1t;
```

¿?

- ¿porqué es particular y compleja?
  - ¿qué información maneja?
  - ¿cómo/cuándo se añade información?
  - ¿cómo/cuándo se consulta?
  - ¿cómo/cuándo se elimina?

```
while a then ...
```

¿?

- ¿los intérpretes la necesitan?
- ¿y los depuradores?
- ¿y los desensambladores?

```
c := i > 1;
```

¿? ¿?

1. Introducción

Tabla de Símbolos: estructura utilizada por el compilador para almacenar información (***atributos***) asociada a los *símbolos* declarados en el programa en compilación.

•Diccionario de un programa

•nombre:

•tipo:

•ámbito:

•dirección:

Puede contener adicionalmente:

- símbolos temporales
- etiquetas
- símbolos predefinidos

•Conceptualmente: *colección de registros*

•Estructura fuertemente influenciada por aspectos sintácticos y semánticos del lenguaje:

- Los tipos disponibles en el lenguaje: determinan el **CONTENIDO** de la tabla.
- Las reglas de ámbito: determina´n la visibilidad de los símbolos, e.g., el **MECANISMO DE ACCESO** a la tabla.

Contenido de la tabla

- **Palabras reservadas**: tienen un significado especial; NO pueden ser redefinidas.

```
program begin end type
var array if ...
```

- **Símbolos predefinidos**: tienen un significado especial, pero pueden ser redefinidos.

- **Literales**, constantes que denotan un valor

- **Símbolos generados** por el compilador:

```
var a: record
         b,c : integer;
       end;
```

  – genera el símbolo `noname1` para el tipo anónimo correspondiente al registro.

- **Símbolos definidos por el programador**
  - **Variables:** tipo, lugar en memoria, ¿valor? ¿referencias?
  - **Tipos de datos:** descripción
  - **Procedimientos y funciones:** dirección, parámetros, tipo de resultado
  - **Parámetros:** tipo de variable, clase de parámetro
  - **Etiquetas:** lugar en el programa

Consulta

Al procesar ..... el compilador .....

- ***declaraciones*** → ***consulta*** la tabla para impedir duplicación ilegal de nombres de símbolos.
- ***sentencias*** → ***consulta*** la tabla para verificar que los símbolos involucrados sean accesibles y se utilicen correctamente.

```
const c = v;
```

- ¿Existe el identificador c?

```
var v : t;
```

- ¿Existe el identificador v?
- ¿Está el tipo `t` definido?

```
v := f (a, b + 1);
```

```
v := e;
```

Actualización

Al procesar ..... el compilador .....

- ***declaraciones*** → ***actualiza*** la tabla para introducir nuevos símbolos.
- ***bloques*** → ***actualiza*** la tabla para modificar la visibilidad de los símbolos.

```
const c = v;
```

– Incluir c con el tipo de v y su valor
– ¿Asignarle una posición en memoria?

```
var v : t;
```

– Incluir v con el tipo t
– Asignarle una posición en memoria

```
end;
```

– Eliminar (ocultar) todos los símbolos del bloque (ámbito) que se cierra.

```
function f (i : integer)
                : integer;
```

– Incluir f (función)
– Abrir un bloque nuevo
– Incluir f (variable *de escritura*)
– Incluir i (parámetro)

```
procedure P (i : integer;
             var b : boolean);
```

– Incluir P (procedimiento)
– Abrir un bloque nuevo
– Incluir i y b (parámetros)

Requerimientos

- ***Rapidez:*** la consulta es la operación más frecuente.

  $O(n)$?

  $O(\log_2(n))$?

  $O(1)$?

- ***Eficiencia en el manejo del espacio:*** se almacena una gran cantidad de información.

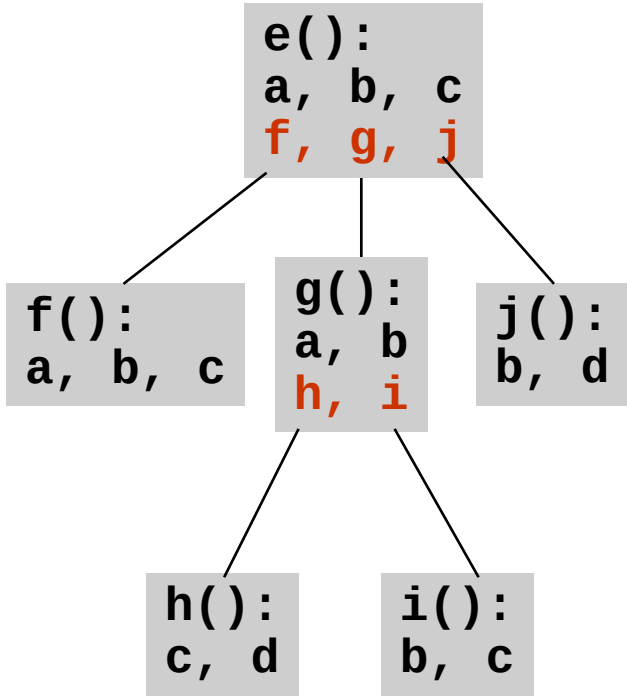
- ***Flexibilidad:*** la posibilidad de definir tipos hace que la declaración de variables pueda ser arbitrariamente compleja.

- ***Facilidad de mantenimiento:*** la eliminación de identificadores debe ser sencilla
  - No es aleatoria

- ***Permitir identificadores duplicados:*** la mayoría de los lenguajes los permiten. Debe estar claro cuáles son accesibles en determinado momento

Requerimientos

```
program e;
var a, b, c : char;
  procedure f;
  var a, b, c : char;
  ...
  procedure g;
  var a, b : char;
    procedure h;
    var  c, d : char;
    ...
    procedure i;
    var b, c : char;
    ...
  ...
  procedure j;
  var b, d : char;
  ...
...
```



Programa en compilación:

```
e()  f()  g()  h()  i()  j()  e()
```

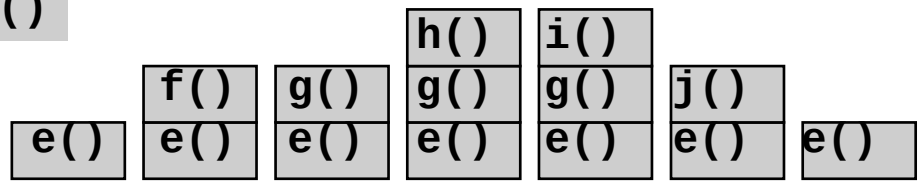


Tabla de símbolos:

2. Nombres

- **Recuerda:** conceptualmente se almacenan registros con los ***nombres*** de símbolos y sus ***atributos***.

- Se precisa un mecanismo de ***almacenamiento y búsqueda*** por nombre.

- **Alternativa 1:** definir el campo nombre como un vector de caracteres.
- FORTRAN IV: limitación al número de caracteres significativos (6). No es así en los lenguajes modernos.
- Variabilidad en la longitud de nombres -> desperdicio de espacio

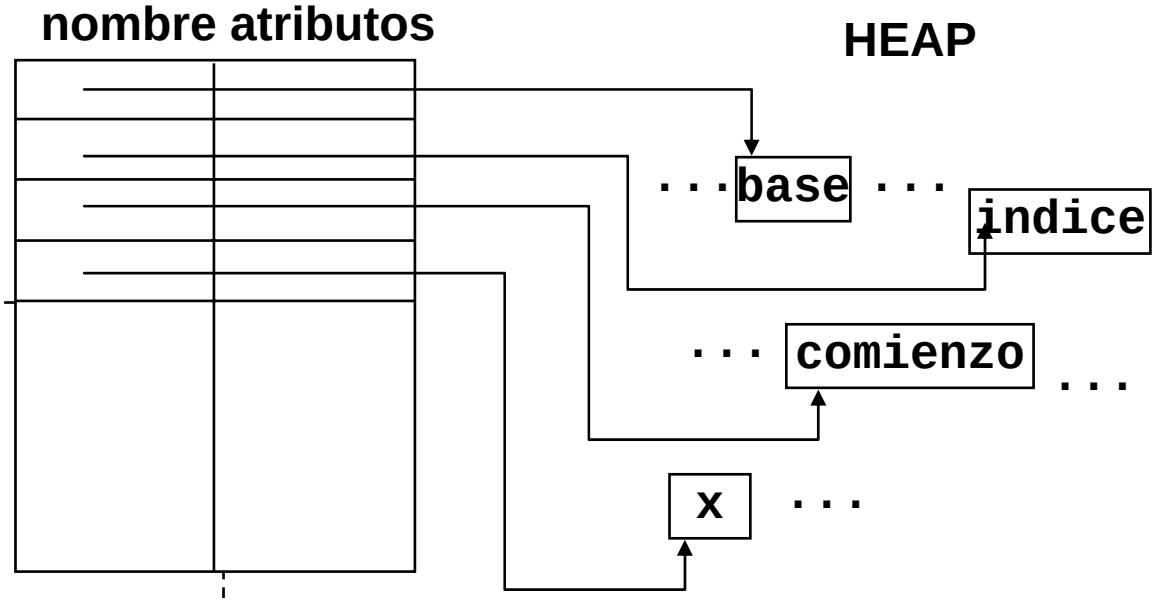
```
struct {
    char nombre[MAX];
    ...
    } entrada_tabla;
```

espacio para identificador más largo posible

nombre	atributos
base	...
indice	
comienzo	
x	
<----->	

Utilizar el Heap

- **Alternativa 2:** definir el campo nombre como un puntero a caracter.

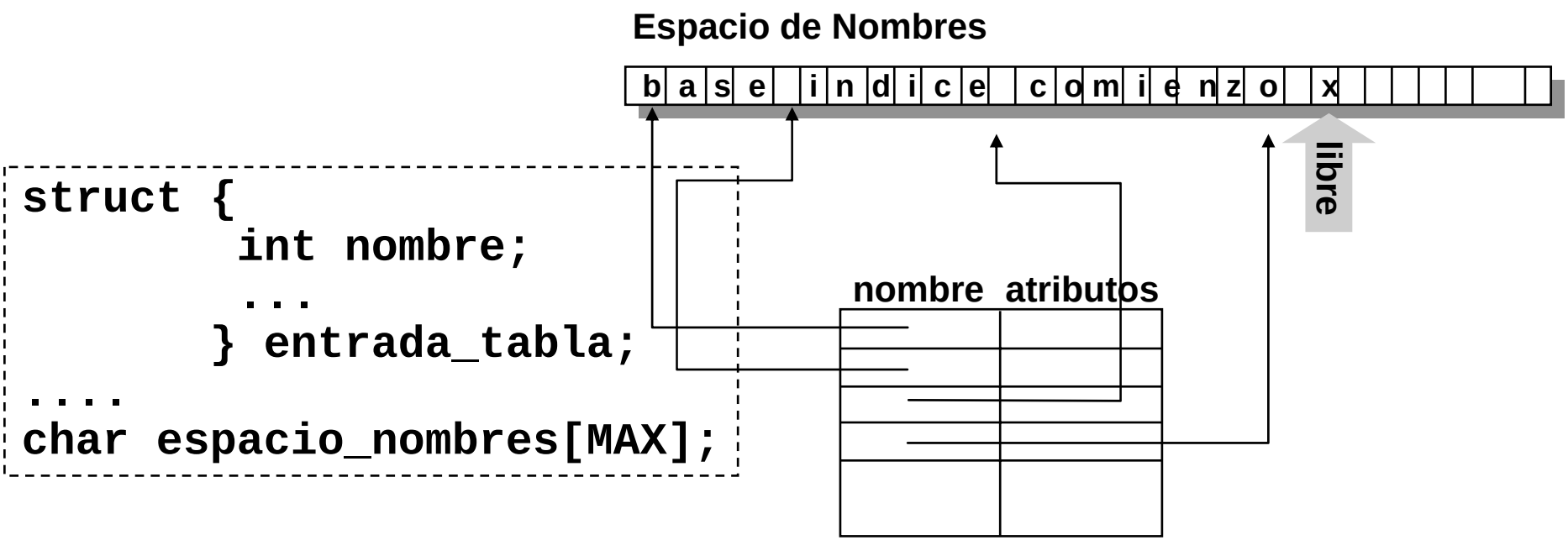



```
struct {
   char *nombre;
   ...
   } entrada_tabla;
...
e->nombre = strdup(nombre);
```

- Obtención de espacio puede ser lenta
- Reutilización depende del mecanismo de recuperación de memoria del HEAP
- Requerimientos para una tabla de símbolos son más simples que los del HEAP

Espacio de nombres

- **Alternativa 3:** definir el campo nombre como un índice en un vector de nombres.



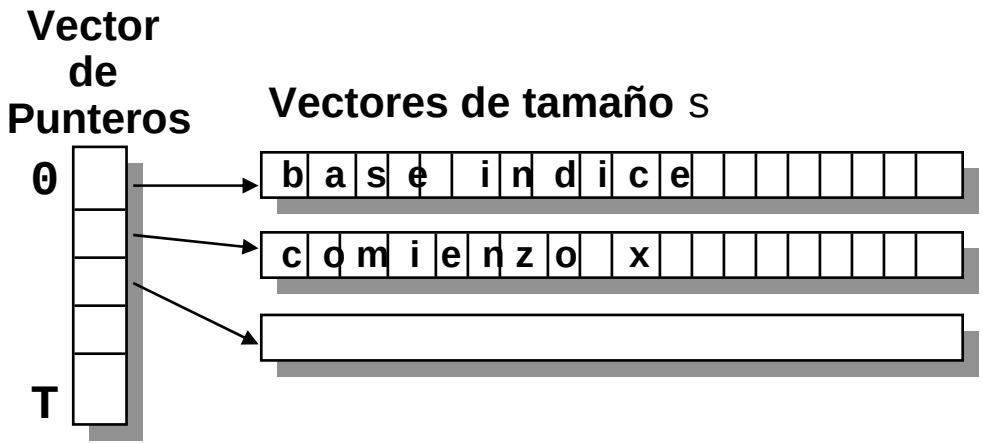
- La administración se hace localmente
- El espacio se reutiliza
- El espacio no es ‘ilimitado’

Espacio de nombres

¿cuál debe ser el tamaño del espacio de nombres?

- Pequeño: puede ser insuficiente
- Grande: puede desperdiciarse espacio

- **Solución:** espacio de nombres segmentado
  - segmento: **`nombre div s`**
  - indice en segmento: **`nombre mod s`**

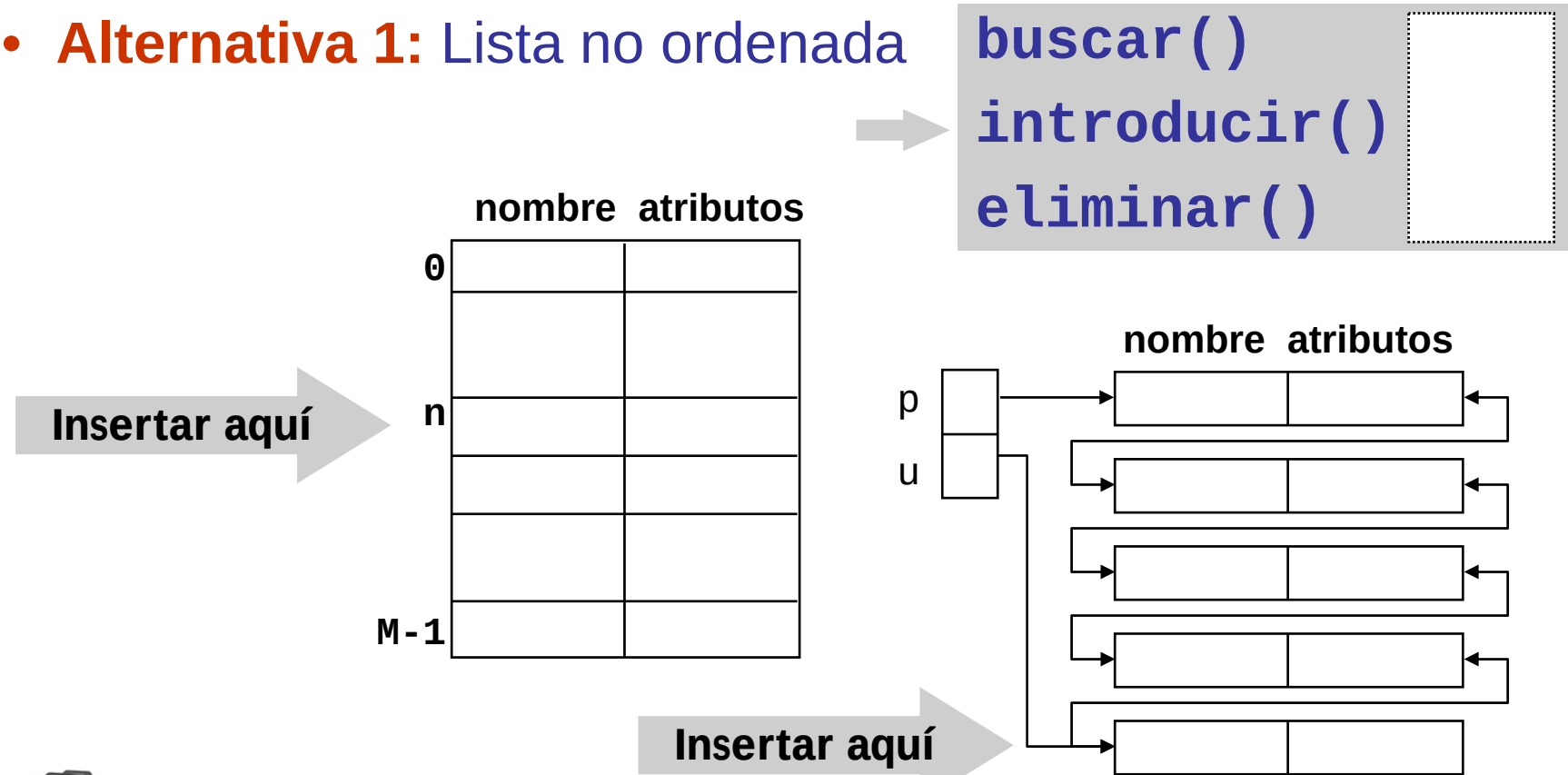



```
nombre = segmento * s + indice
```

- Tamaño limitado por el tamaño del vector de punteros (T=50 punteros de 1024 caracteres = espacio de 50k)

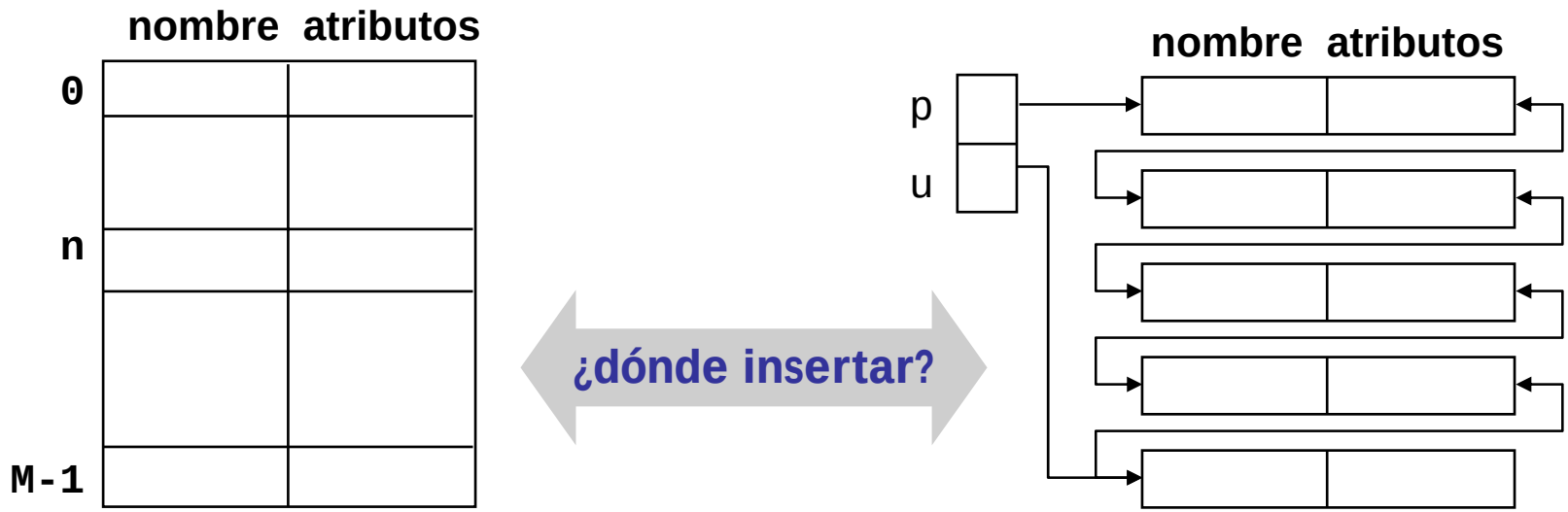
3. Organización

- Tres operaciones fundamentales:
  - `buscar()`
  - `introducir()`
  - `eliminar()`
- **Alternativa 1:** Lista no ordenada



3. Organización

- **Alternativa 2**: Lista ordenada por nombre



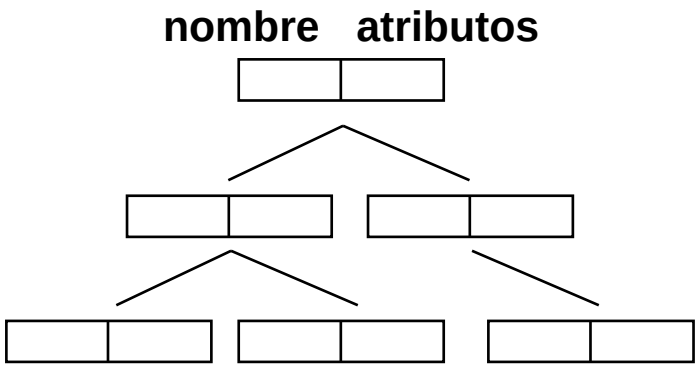

```
buscar()
introducir()
eliminar()
```

¡Se pierde el orden de inserción!

3. Organización

- **Alternativa 3:** Árboles binarios

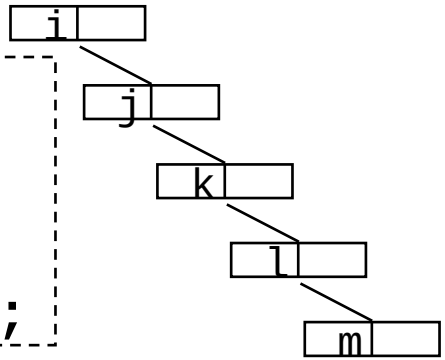
¡Sólo si está balanceado!



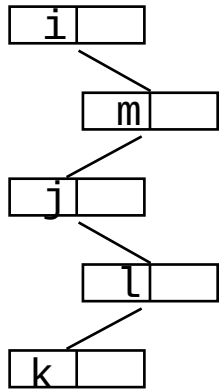

```
buscar()
introducir()
eliminar()
```

- En el caso más desfavorable, la complejidad de las operaciones es igual a la de la lista ordenada.

```
var i,
    j,
    k,
    l,
    m : integer;
```



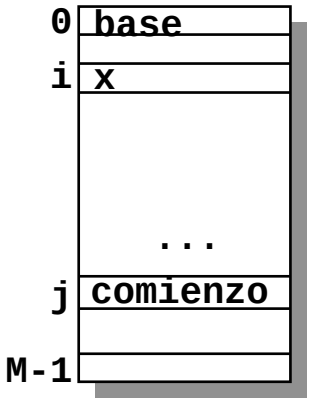

```
var i,
    m,
    j,
    l,
    k: integer;
```



- No se puede garantizar que el árbol sea balanceado (los nombres no son aleatorios).

3. Organización

- **Alternativa 4: tablas de dispersión**. Mecanismo para distribuir aleatoriamente un número arbitrario de ítems en un conjunto finito de clases.




```
h('base') = 0
h('x') = i
h('comienzo') = j
```

¿h?

- Función de dispersión **h**: asocia una cadena de caracteres con un código de dispersión = índice en la tabla.

- Colisiones: dos cadenas diferentes pueden estar asociadas al mismo índice.

¿cómo manejarlas?

- Con una función de dispersión adecuada, y un manejo de colisiones eficiente, se puede **`buscar()`** en tiempo **constante**.

La función de dispersión h

- Características deseables:
  - **`h(s)`** debe depender sólo de **`s`**
  - **Eficiencia**: debe ser fácil y rápida del calcular
  - **Eficacia**: debe producir listas de colisiones pequeñas
    - » **Uniforme**: todos los índices asignados con igual probabilidad
    - » ***Randomizante***: nombres similares a direcciones diferentes
- Paradoja del cumpleaños:
  - Tenemos **`N`** nombres, y una tabla de tamaño **`M`**
  - Sea **`h`** uniforme y *randomizante*. Inserciones antes de una colisión:

`M`	**`sqrt(pi M/2)`**
10	4
365	24
1.000	40
10.000	125
100.000	396

Números aleatorios entre 0 y 100:

84 35 45 32 89 1 58 16 38 69 5 90 16 53 61 ...

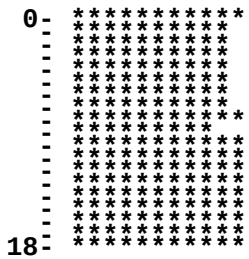
Colisión al 13avo.

Ejemplos

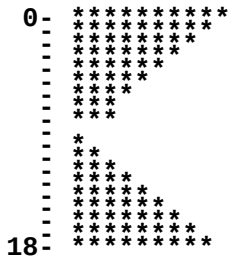
Método de la división

```
function hash_add(s : string; M : integer) : integer;
      var k : integer;
begin
    k := 0;
    for i := 1 to length(s) do k := k + ord(s[i]);
    hash_add := k mod M;
end;
```

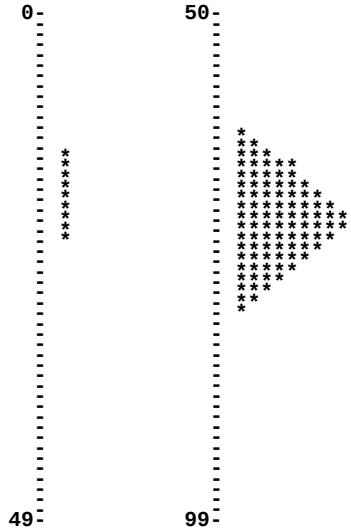
- Tamaño = 19, ids A0...A199



- Tamaño = 19, ids A0...A99



- Tamaño = 100, ids A0...A99



No maneja anagramas

Ejemplos

```
int hash_horner(char *s, int M)
{
  int h, a = 256;
  for (h=0; *s != '\0'; s++)  h = (a*h + *s) % M;

  return h;
}
```

bcba	47	ccad	1	baca	26	abad	4
bddc	43	bdac	83	dbcb	24	cada	85
dabc	85	dabb	84	dbab	17	dabd	86
dbdb	78	dcbd	60	dbdd	80		
babb	74	bccc	2	addd	39		
bcbd	50	adbc	31	bcda	55		

Test: **'abcd'**

256*(256*(256*97+98)+99)+100

mod después de cada multiplicación:

```
256*97+98  = 24930 % 101 = 84
256*84+99  = 21603 % 101 = 90
256*90+100 = 23140 % 101 = 11
```

Colisión al 21 (desviación estándar grande).

Más ejemplos

```
unsigned hash_pjw (unsigned char *s)
{
    unsigned i, hash_val = 0;
    for (; *s; ++s) {
        hash_val = (hash_val << 2) + *s;
        if (i = hash_val & 0x3fff)
            hash_val = (hash_val ^(i >> 12)) & ~0x3fff;
    }
    return hash_val;
}
```

- Longitud media igual a **hash_add**
- 9% más lenta que **hash_add** [Holub 90].

```
function uw_hash (s : string; M : integer) : integer;
   var l : integer;
begin
   l := length(s);
   uw_hash := (ord(s[1]) * ord(s[l])) mod M;
end;
```

- UW-PASCAL: sencilla, funciona razonablemente; es preferible utilizar todos los caracteres.

[Pearson, 90]

- Específica para cadenas de texto de longitud variable

```
h = 0 ;
for i in 1..n loop
    h := T[h xor C[i]];
end loop ;
return h
```

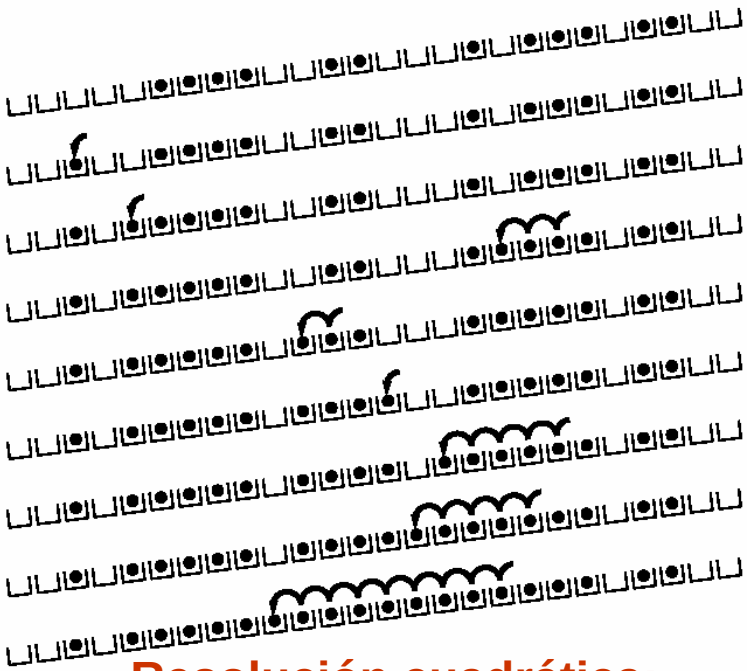
- **n**: longitud de la cadena
- **C[i]**: i-ésimo caracter
- **T**: números aleatorios (una permutación de **0..255**).
- Ventajas:
  - Cambios pequeños en las cadenas (cadenas similares) resultan en cambios grandes en el resultado (proviene de la criptografía).
  - Maneja anagramas
- **Otros:** sumar impares, mayor código ASCII

Manejo de colisiones

- **Mantener `N < M`:** si **`h(s)mod M`** está ocupada intentar:
  - » **Resolución lineal:**

`(h(s) + 1) mod M, (h(s) + 2) mod M`, etc.

clustering primario



Lo que va mal tiende a empeorar:

- Se forman largas cadenas, que tienden a crecer más
- El costo promedio de **`buscar()`** se acerca a M a medida que la tabla se llena
- Demasiado lento si la tabla está al 70%-80%

  - » **Resolución cuadrática:**

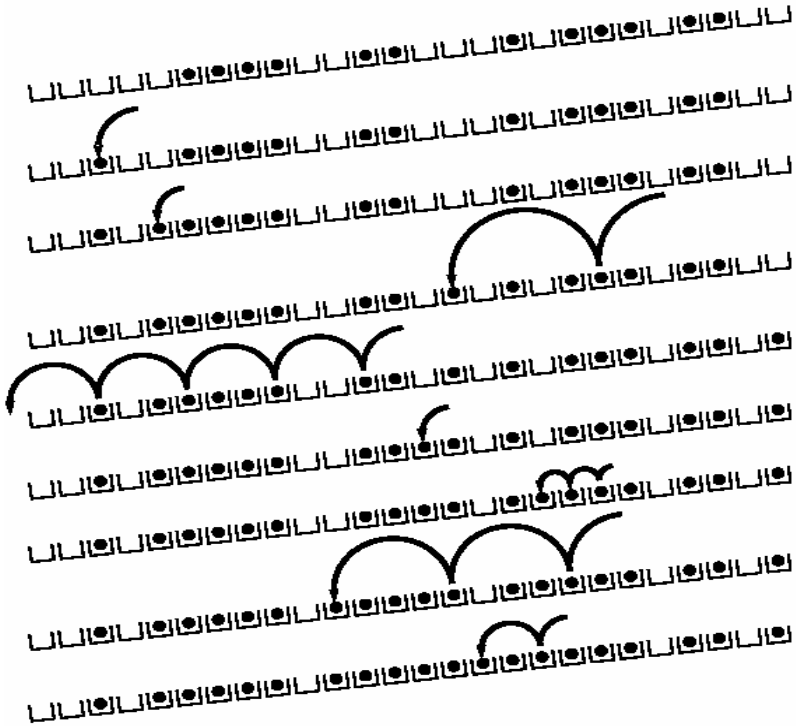
`(h(s) + 1^2) mod M, (h(s) + 2^2) mod M`, etc.

- Tiene un problema parecido (*clustering* secundario)

Manejo de colisiones

» **Hashing doble:** otra función **h'** determina los intervalos de los saltos

```
 h(s) mod M,
(h(s) + h'(s)) mod M,
(h(s) + 2*h'(s)) mod M,
etc.
```



- Análisis muy complejo
- No es demasiado lento hasta que la tabla se ocupa al 90%-95%

Escoger el valor de M

- Evitar potencias de 2:

  $k \bmod 2^b$ selecciona los b bits más bajos de k

  – Ignora parte de la información

  – Si la función no es uniforme el los bits bajos, genera colisiones innecesariamente

- Hashing doble: **$h'(s)$ y M deben ser primos relativos**

  - Puede hacer imposible encontrar un sitio libre en una tabla no llena!

  - La experiencia aconseja escoger números primos cercanos a potencias de 2:

Test:

```
h = 10, h' = 3, M = 9
1 4 7 1 4 7 1 4 7 1 4 7 ...

h = 10, h' = 3, M = 10
0 3 6 9 2 5 8 1 4 7 0 3 ...
```

Tabla de ~4000 elementos:

$m = 4093$

Mayor n. primo $\leq 4096 = 2^{12}$

Manejo de colisiones

- **Permitir N >> M (alternativa más usual)**: Encadenamiento de las colisiones (tablas abiertas).
  - No falla cuando se llena la tabla
  - Ocupación mínima de espacio en la tabla
  - Función uniforme → acceso ≈ constante

¿las colisiones pueden ser un árbol binario?

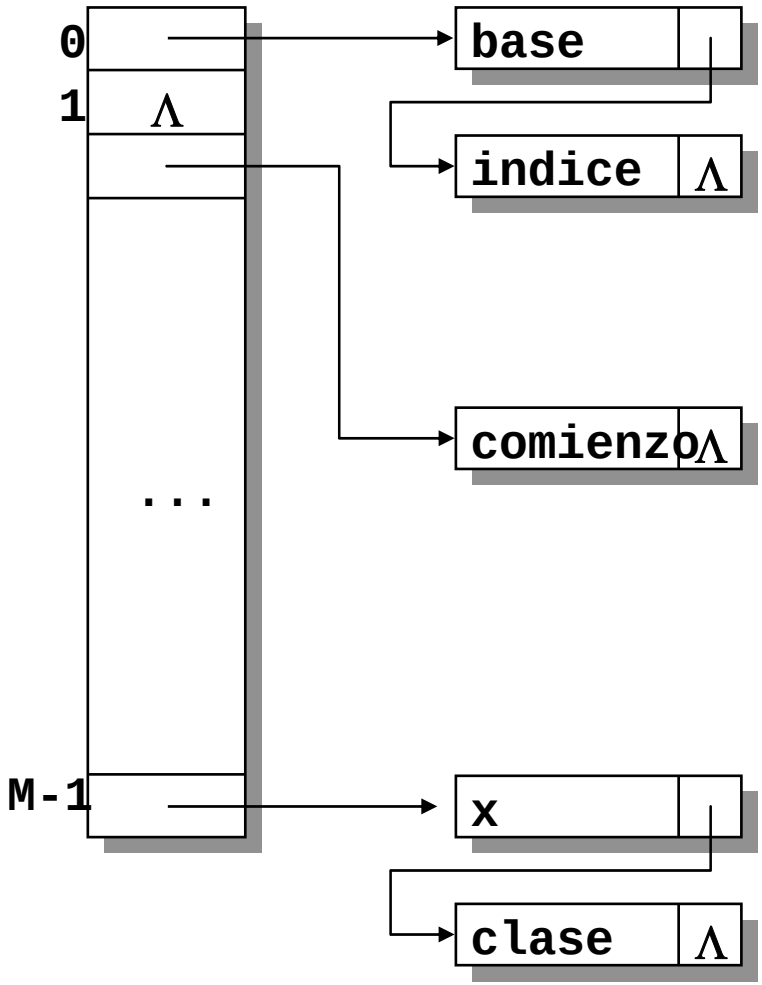
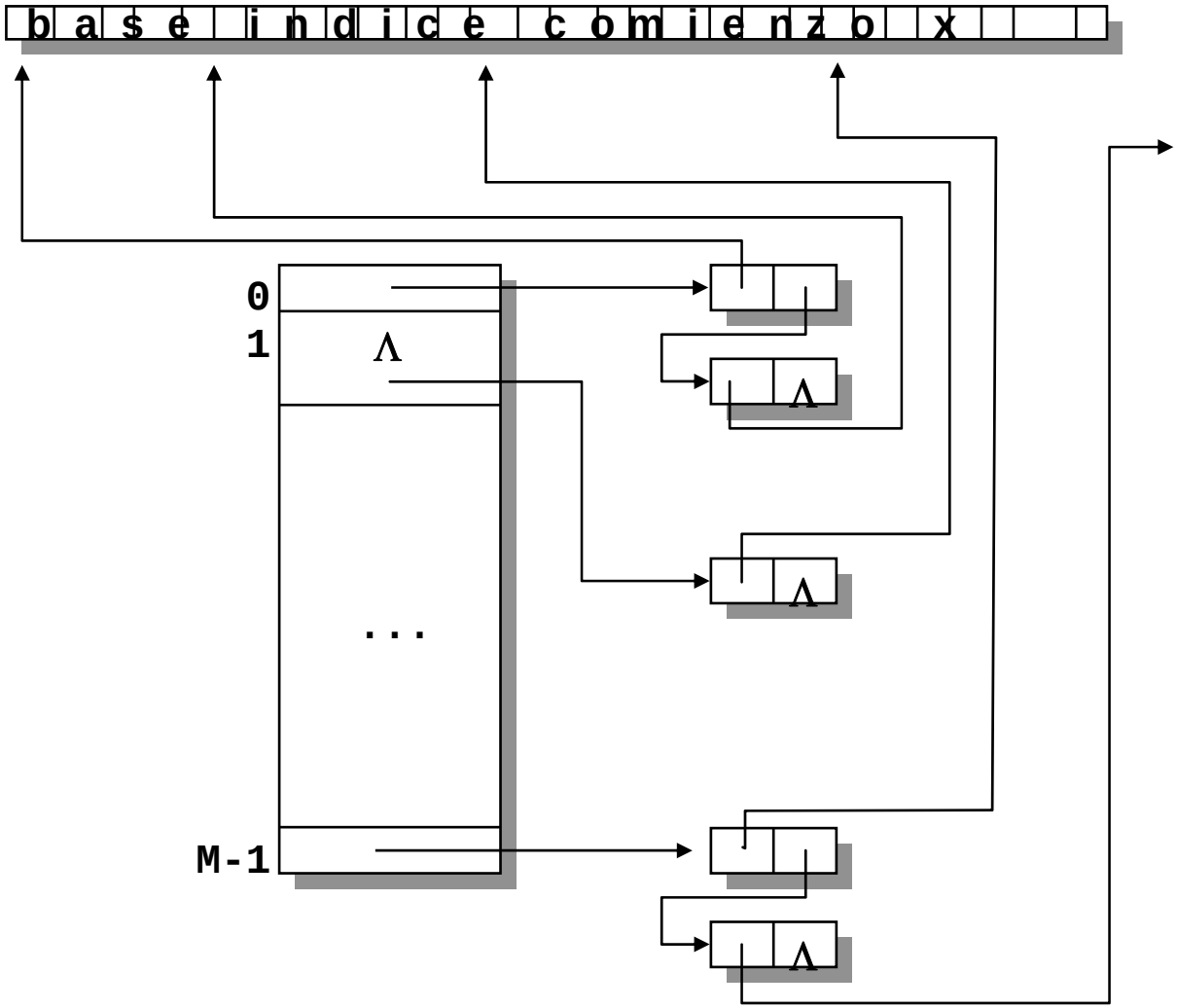


Tabla de dispersión y nombres (son ortogonales)



4. Lenguajes de Bloques

- Block Structured Languajes (Algol 60):
  - Cada línea de un programa pertenece a una serie de bloques (ámbitos) anidados *(**abiertos**)*
  - El más interno es el ***bloque actual***
  - Los bloques que no contienen la línea están ***cerrados***

- Reglas de visibilidad:
  - El bloque en el que está declarado un objeto es su ***ámbito***
  - En cualquier línea del programa, sólo los objetos declarados en son accesibles.
  - Sólo puede haber nuevas declaraciones en el bloque actual.
  - Si un objeto está declarado en más de un bloque abierto, se utiliza la definición del bloque más interno (la más).

- Implicaciones:
  - El último bloque en abrir es el en cerrar.
  - Una vez cerrado un bloque, no se vuelve a abrir.
  - Al cerrar un bloque, todas sus declaraciones son inaccesibles.

Ejemplos: Pascal

```
program pr;

var a : char;

function f (i : integer): integer;
  var d : real;
begin
     ..
end;                                   1

procedure  p (var j: char)
  var i, f : real;
begin

       ..
       procedure q ;
          var i, c : integer;
       begin

         f = i*d;
       end                             2
end;                                   1

begin
...
end.                                   0
```

¿?

Ejemplos: C

```
char a;
 int f (int i, int j, char* c)
 {
     double d;
     ..
 }                                  1

 void g (char j, int c)
 {
     double i, f;
     static int d;
     {
       int i, j, c;
       f = i*a;
     }                              2
 }                                  1
main ()
{
...
}                                   0
```

¿?

4. Lenguajes de Bloques

- Consideramos ámbitos ***estáticos***: al compilar, puedo determinar a qué objeto se refiere cada instrucción.
- Ámbitos ***dinámicos***: esto se determina durante la ejecución.

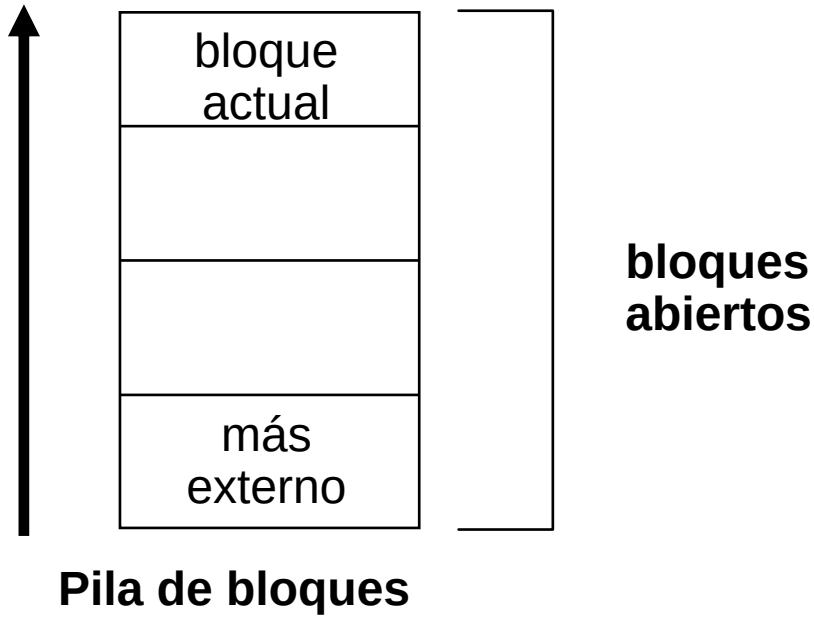
```
a : integer                    -- global declaration

procedure first
    a := 1

procedure second
    a : integer                -- local declaration
    first ()

a := 2
if read_integer() > 0
   second ()
else
   first ()

write_integer(a)
```

¿qué escribe este programa en cada caso?

4. Lenguajes de Bloques

- **Alternativa 1**: Una tabla por cada ámbito:

- Abrir un bloque: **`apilar()`**
- Cerrar un bloque: **`desapilar()`**
- **`buscar()`:** se busca desde el tope hacia abajo.



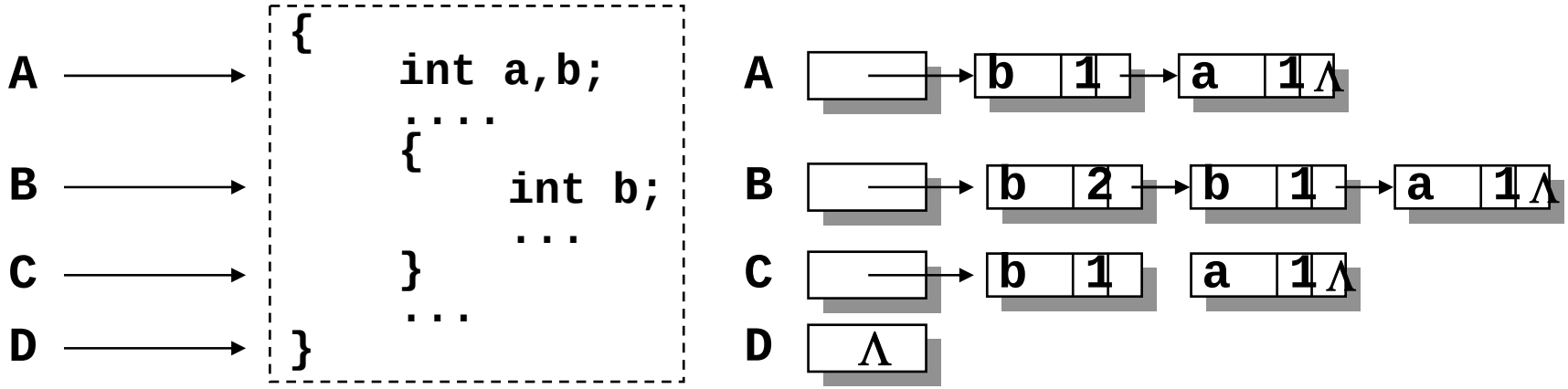
- **Ventaja:** la operación de **`desapilar()`** es muy eficiente.

- **Desventaja 1:** la eficiencia de **`buscar()`** depende de la profundidad del anidamiento.

- **Desventaja 2:** c/bloque requiere una tabla de dispersión; desperdicio de espacio.

4. Lenguajes de Bloques

- **Alternativa 2**: una sola tabla.
  - A c/objeto se le asocia un número de bloque diferente.
  - Nombres nuevos al comienzo de la lista de dispersión.
  - Al cerrar un bloque, se eliminan todos los objetos del bloque a cerrar.

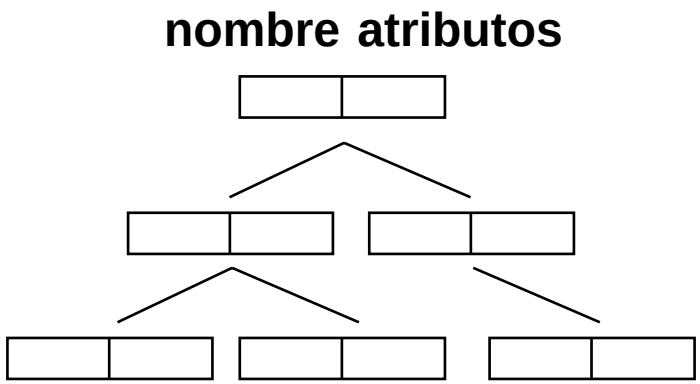


- **Ventaja:** la búsqueda es más eficiente

- **Desventaja:** la eliminación es menos eficiente
  - En casos en los que se guarda la tabla de símbolos, esta operación no se usa.

Alternativa 2: una sola tabla

- Esta alternativa no es tan apropiada si las colisiones se almacenan en árboles binarios:



- La inserción **siempre** se hace en las hojas.
- La búsqueda también deberá ir hasta las hojas para determinar cuál es la definición más interna un objeto.
- Cerrar un bloque implica recorrer todo el árbol.

¡Se pierde el orden de inserción!

Resumen

- Componentes de una tabla de símbolos:
  1. **Espacio de nombres**: se almacenan los nombres de los símbolos introducidos en el programa.
  2. **Mecanismo de acceso**: se utiliza para comenzar la búsqueda de un símbolo.
     - » Lista encadenadas
     - » Árboles binarios
     - » *Tablas de dispersión*
  3. **Tabla de atributos**: se almacenan los atributos asociados a un símbolo.

¿ ?

- Debe respetarse la estructuración de bloques / ámbitos definidos en el programa.
  - **Pilas de bloques**: actualización más eficiente, búsqueda poco eficiente.
  - **Una tabla para todos los bloques**: búsqueda muy eficiente, actualización menos eficiente.

5. Perspectiva

- **Registros y Campos:** En C, PASCAL y ADA el nombre del campo sólo debe ser único dentro de la definición del registro:

```
type r1 = record
        a : integer;
        b : real;
      end;

      r2 = record
        a : char;
        b : boolean;
      end;
```

```
type a = record
        a : integer;
        b : record
          a : char;
          b : boolean;
        end;
      end;
```

- Aumenta la legibilidad, y facilidad de programación.

¿Cómo implementar esto en la tabla de símbolos evitando ambigüedades?

5. Perspectiva

- **Referencias hacia adelante:** en algunos lenguajes se permite usar nombres antes de declararlos:

- Pascal: registros

```
type asignatura = record
   profesor = ^empleado;
   delegado = ^alumno;
   ...
end;

empleado = record
   ...
end;
```

- Se permite

```
alumno = record
   cursa = ^asignatura;
   ...
end;
```

- ADA: etiquetas

```
<<etiqueta>>
...
...
goto etiqueta;
```

- Se mantiene para dar soporte a generadores automáticos de programas
- En algunos casos evitan anidamientos excesivos

5. Perspectiva

- **Visibilidad**: Las reglas de visibilidad pueden ser diferentes en situaciones especiales:
  - PASCAL: Los nombres de los campos de los registros no son visibles sin el nombre del registro.

```
type r = record
        n : integer;
        c : char;
        b : boolean;
      end;

var v : r;
.....
   n := 0;
```

  - ADA: prefijos

```
procedure P;
var X : real;
  procedure Q;
  var X : real;
    begin
    X := P.X;
    ...
```

5. Perspectiva

- **Alteración de reglas de visibilidad**: Las reglas de visibilidad pueden ser alteradas por el programador:
  - PASCAL: sentencia `with`.

```
type r = record
       n : integer;
       c : char;
       b : boolean;
     end;

var v : r;
.....
   with v do
        n := 0;
```

- Problemas:

```
var v1, v2 : r;
.....
   with v1, v2 do
        n := 0;
```

- Resuelto en Modula-3:

```
WITH e = loquesea,
     f = loquesea DO
e.f1 = f.f2;
END;
```

5. Perspectiva

- **Compilación separada**: un programa se construye con módulos compilados separadamente.
  - **ADA:** se basa en el concepto de *biblioteca*

p.ads

```
package P is
   type t is private;

   procedure q(.....);
   function f(.....) return t1;
     .....
end P;
```

q.ads

```
with P:
```

  - **C:** no existe el concepto de biblioteca en compilación

p.h

```
typedef ..... t;

void q(.....);
t1 f(.....);
```

q.c

```
#include "p.h"
```

5. Perspectiva

- **Sobrecarga**: el mismo símbolo tiene significados diferentes dependiendo del contexto.
  - PASCAL: operadores aritméticos.

```
var i, j : integer;
    r, s : real;
....
j := i * 2;
s := r * 2.0;
```

  - PASCAL: funciones

```
function f (n : integer)
                    : integer;
begin
    if n > 1 then
        f := n * f (n-1)
    else
        f := 1
end;
```

  - ADA: Concepto de sobrecarga generalizado.

```
function "+" (x, y : complejo)
              return complejo is ...
function "+" (u, v : polar)
              return polar is ...
```

6. tabla.c y tabla.h

- Implementación de una tabla de símbolos en forma de tabla de colisiones abierta.

```
#include <listas.h>

#define TAMANO  7
typedef struct {
    char *nombre;
    int nivel;
    TIPO_SIMBOLO tipo;
    TIPO_VARIABLE variable;
    int dir;
    ...
} SIMBOLO;

typedef LISTA
       TABLA_SIMBOLOS[TAMANO];
```

```
SIMBOLO *buscar_simbolo(
         TABLA_SIMBOLOS tabla,
         char *nombre)

/* NULL si el nombre no aparece
en la tabla, dlc puntero al
 simbolo mas reciente */
```

```
SIMBOLO *introducir_variable (TABLA_SIMBOLOS tabla,
                              char *nombre,
                              TIPO_VARIABLE variable,
                              int nivel, int dir);

/* NULL si el nombre ya aparece en la tabla con igual nivel, dlc
   puntero a simbolo creado con esos datos */
```

7. Ejercicios

1. Considera la posibilidad de que tu compilador de Pascual detecte **variables utilizadas sin tener valor asignado**.

```
programa p;
entero i;

principio
     escribir(i);
fin
```

¿cómo se asigna valor a una variable?

¿cuándo se consulta el valor de una variable?

Tarea a)

Explica qué habría que modificar en la tabla de símbolos (tanto en la definición del tipo **SIMBOLO** como de las funciones asociadas) para detectar estas situaciones.

Tarea b)

Indica qué producciones habría que modificar, y cómo, para implementar esta verificación.

```
leer: tLEER '(' lista_asignables ')' ';'
;

lista_asignables: tIDENTIFICADOR
                  {




                  }
|                 lista_asignables ',' tIDENTIFICADOR
;
```

Tarea b)

Indica qué producciones habría que modificar, y cómo, para implementar esta verificación.

```
expresion:
...

factor: tIDENTIFICADOR
        {




        }
```

Tarea c)

Explica si tu solución tiene limitaciones, es decir, si en algún caso puede NO detecta una situación de este tipo.

Inicializadores

2. Considera la posibilidad de utilizar **inicializadores** para variables simples en Pascual, ilustrada en el siguiente ejemplo:

```
programa p;

var n = 1, c = "c", b = true;
.....
accion q (entero E i; caracter ES d);
var j = 2*i+1, g = entacar(caraent(d)+1), f = not b;
.....
```

- Como puedes ver, en la sintaxis propuesta está explícito el valor inicial, e implícito el tipo de variable.

Tarea a)

Escribe las producciones que especifican sintácticamente este tipo de declaración de variables.

Tarea b)

Completa las producciones anteriores para incluir las verificaciones semánticas y actualizaciones de tabla de símbolos que sean pertinentes. Define los atributos que requieras para cada símbolo.

Funciones

3. Considera el siguiente trozo de programa Pascal:

```
function f (......) : .....;
     function g (.......) : .....;
     begin
          .....
          f := .....;
          .....
     end
begin
     .....
end
```

- **TAREA a)** Desde el punto de vista del compilador, ¿es esto aceptable? Explica tu respuesta.
- **TAREA b)** Explica si esta situación se da o no en C.

Ámbitos

4. En un lenguaje con un solo ámbito, variables de un tipo, y sin declaraciones, ¿puede el analizador léxico manejar la tabla de símbolos?

```
10 REM programa simple en BASIC 4
20 REM
30 z = 0
40 IF a > 0 GOTO 60
50    z = 1
60 PRINT z, a
```