Lenguajes de Programación I
Apunte Final

Tabla de contenido

Clase 1. BNF (Backus Normal Form)

Introducción

El rol de las variables en el lenguaje de programación (variables, constantes, palabras reservadas, parámetros, tipos clases, etc.) es el rol que tienen las palabras en el lenguaje natural.

Una regla gramatical sería algo así como una declaración, se escribe: primero el tipo luego las variables separadas por coma, y finalmente punto y coma.

<tipo> <variable 1>, <variable 2>,…, <variable n>;

La regla léxicas dicen como se escribe el tipo y como se escriben las variables.

Las reglas sintácticas dicen como se combina el tipo, las variables y el punto y coma para hacer una declaración.

O sea las reglas gramaticales son más amplias. Abarcan aspectos más grandes del programa.

Esto es una meta visión del problema, pero ocurre que para hacer un compilador utilizando estas especificaciones, la probabilidad de hacerlo bien, va a ser muy baja y además el trabajo va a ser muy duro. Entonces desde ya hace mucho tiempo se empezó a procurar de escribir las reglas formalmente, o formalizar estas reglas.

O sea tener un metalenguaje, o sea otro lenguaje cuyo objetivo es describir la sintaxis del lenguaje. O sea un metalenguaje que describa la sintaxis del texto.

A fines del año 1956 John W. Backus, escribió el lenguaje Fortran (**For**mula **Tran**slating System). Fue el primer lenguaje de programación. El primero con una estructura separada de la maquina, que ya no había necesidad de conocer la máquina para hacer un programa. Fue una inmensa novedad que después se fue diluyendo y ahora es un lenguaje casi extinguido, se usa pero muy poco.

Backus, tuvo un problema, tuvo que hacer el compilador de Fortran y para hacer el compilador de Fortran creó un lenguaje para describir la sintaxis de Fortran. Ese lenguaje, él no le dio nombre, él lo usó, pero otras personas lo llamaron BNF (Backus Normal Form), algunas bibliografías lo llaman (Backus-Naur Form) debido a que Naur trabajaba con Backus. Hoy gran parte del dominio de la computación utiliza BNF para describir la sintaxis del lenguaje de programación.

Un **identificador** es un término genérico para referirse a un nombre con ciertas propiedades. Mientras que, una **variable** es un identificador que ha sido declarado con un tipo. El nombre de una función también es un identificador, el nombre de un parámetro, el nombre de un tipo (definido por el usuario), también son identificadores. O sea que identificador son todos los nombres aislados utilizados para un propósito en un programa. Variables son aquellos identificadores que se muestran como variables.

Elementos utilizados para definir un lenguaje en BNF

Flecha de Definición (→): la fecha a la derecha significa “está definido como”. Se usa de la siguiente manera: A → B donde A es el objeto que se está definiendo, y B es como se define ese objeto.

Entidades No Terminales < NT >: son caracteres que encierran entidades No Terminales. Por ejemplo, esto es una entidad no terminal: <ASIGNACION>. Todas las entidades no terminales necesitan ser definidas por medio de la flecha de definición vista anteriormente.

Entidades Terminales: Son los símbolos del alfabeto. No deben ser definidos por medio del lenguaje BNF. Ejemplos de elementos terminales son el punto, la coma, las letras, los dígitos, etc.

Podemos usar el | (pipe) para simplificar varias reglas que definen a la misma entidad no terminal.

Ejemplo de una sentencia BNF:

<DIGITO> → 0 | ... | 9
<LETRA> → a | ... | z
<VARIABLE> → <LETRA> | <VARIABLE><LETRA>|<VARIABLE><DIGITO>

Con estas reglas lo que queremos hacer es definir que una variable puede ser representada por una seguidilla de letras y dígitos pero con la restricción de que no puede comenzar con un dígito.

La primer regla define a la entidad no terminal DIGITO como cualquiera de los dígitos del 0 al 9 (estas son entidades terminales y no necesitamos definirlas).

La segunda regla define a la entidad no terminal LETRA como cualquiera de las letras de la “a” a la “z” minúsculas (estas son entidades terminales y no necesitamos definirlas).

Con la última regla (que en realidad son tres reglas separadas por pipes que definen la misma cosa) definimos a la entidad no Terminal VARIABLE. En el primer caso nos dice que una variable puede ser simplemente una letra. En los otros dos casos nos dice que una variable puede ser lo que conocemos como variable y además una letra o un dígito final.

Lenguajes como C usan aproximadamente 250 reglas.

Recursividad de la gramática

Es la particularidad más importante de la gramática. Ejemplo de líos con las recursividades a derecha o a izquierda:

Gramática recursiva a izquierda:
R1: <EXPRESION> → <EXPRESION> + <TERMINO>
R2: <EXPRESION> → <EXPRESION> - <TERMINO>
R3: <EXPRESION> → <TERMINO>
R4: <TERMINO> → <TERMINO> * <FACTOR>
R5: <TERMINO> → <TERMINO> / <FACTOR>
R6: <TERMINO> → <FACTOR>
R7: <FACTOR> → <IDENTIFICADOR>
R8: <FACTOR> → <CTE>

Gramática recursiva a derecha:
R1: <EXPRESION> → <TERMINO> + <EXPRESION>
R2: <EXPRESION> → <TERMINO> - <EXPRESION>
R3: <EXPRESION> → <TERMINO>
R4: <TERMINO> → <FACTOR> * <TERMINO>
R5: <TERMINO> → <FACTOR> / <TERMINO>
R6: <TERMINO> → <FACTOR>
R7: <FACTOR> → <IDENTIFICADOR>

R8: <FACTOR> → <CTE>

Son gramáticas que aceptan el mismo lenguaje pero los efectos semánticos no son iguales.

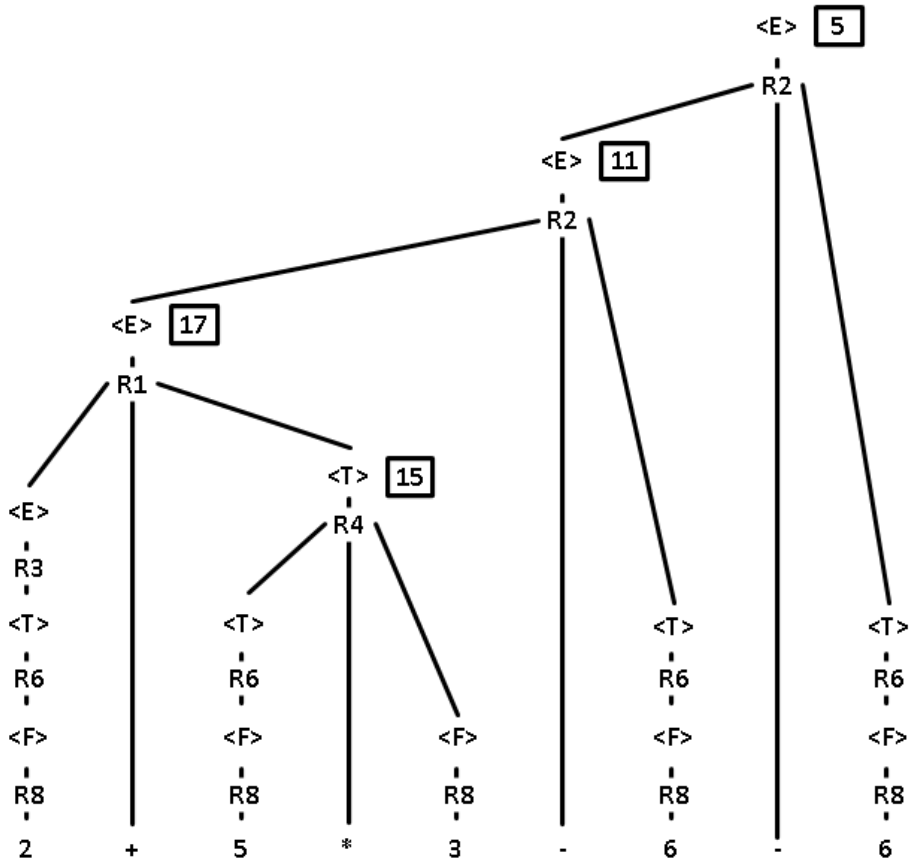
Si el programa está mal escrito la gramática me lo tiene que decir de alguna manera.

Las gramáticas siempre tienen un símbolo más importante, en el ejemplo el símbolo más importante es <EXPRESION>.

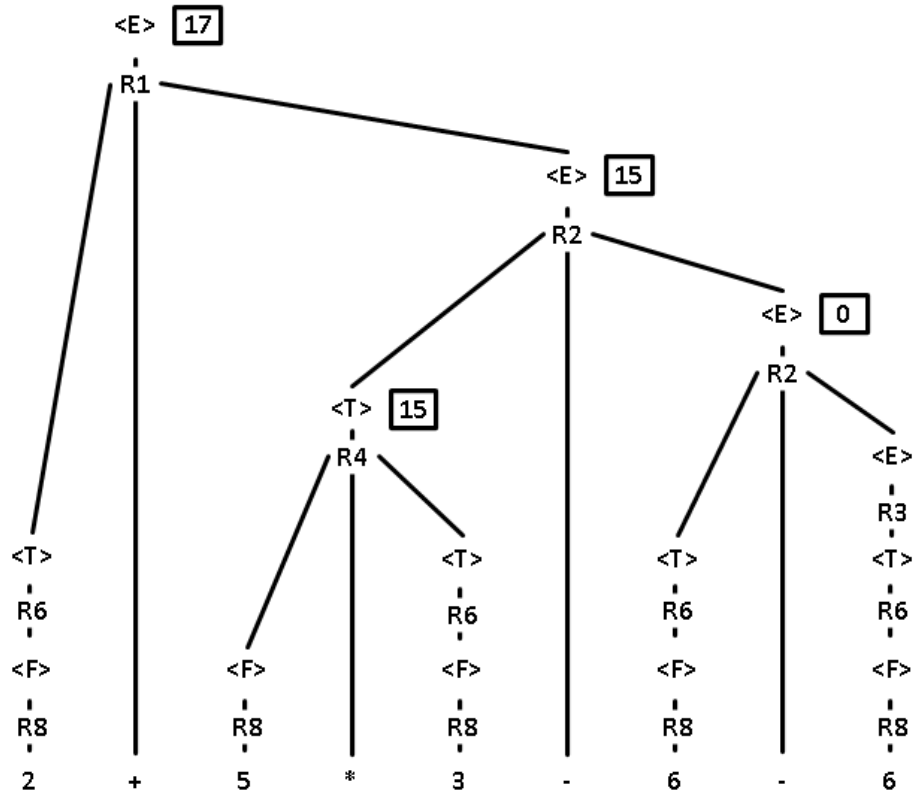
El **árbol de Parsing**: es una estructura que tiene en la cúspide <EXPRESION>, y en la base el programa o en este caso la <EXPRESION> concreta. Es la forma de verificar un programa o una expresión o cualquier cosa que se escriba correspondiendo la gramática.

Ejemplo: $2 + 5 * 3 - 6 - 6$

Árbol de Parsing de la gramática recursiva a izquierda:



Árbol de Parsing de la gramática recursiva a derecha:



Si se puede construir el árbol de Parsing, entonces el programa está bien escrito. Esta es la forma que tienen los compiladores de verificar el programa, o sea que cada vez que se compila un programa, se crea el árbol de Parsing. Un detalle importante es que si la gramática no es ambigua el árbol de Parsin es único y viceversa.

Si bien en el ejemplo, los arboles pueden construirse vemos que el resultado final de cada gramática utilizada es diferente. Esto se debe a que la **recursividad influye en la asociatividad** de las operaciones, de forma que al utilizar operadores no asociativos como la resta o división, la gramática recursiva a izquierda y a derecha presentan diferencias.

En las gramáticas recursivas a izquierda, los arboles quedan recostados a izquierda. Y con gramáticas recursivas a derechas los arboles quedan recostados a derecha.

Además se pude apreciar otra propiedad de la gramática, que es la **precedencia de operadores**. Ambas gramáticas tienen precedencia de producto y división sobre suma y resta.

Por cada precedencia de operadores que se quiera, se necesita un no terminal adicional.

Hay dos tipos distintos de Parsing:

- **Parsing ascendente:** comienza del programa y llega a la hipótesis, (regla de inicio)
- **Parsing descendente:** reemplaza todo no terminal por una de sus definiciones hasta llegar a formar el programa.

Clase 2. Semántica del lenguaje de programación

Introducción

El elemento más apropiado por el cual combine comenzar para hablar de semántica es por la noción de binding o ligadura.

Binding: es el estudio del momento exacto en que se conoce una **propiedad** de un **elemento del lenguaje**.

Hay muchos elementos del lenguaje pero la más útil para empezar a comprender la semántica es arrancar por las variables. Entonces ahora la definición seria.

Binding: es el estudio del momento exacto en que se conoce una **propiedad** de una **variable**.

Para hacerlo más preciso si la propiedad que estudio es el tipo. Entonces ahora esta definición es mucho más concreta, mucho más especifica.

Binding: es el estudio del momento exacto en que se conoce una **tipo** de una **variable**.

Los lenguajes se diferencian enormemente por cuando se conoce el tipo de una variable. Por ejemplo, un lenguaje como C, basta leer el texto del programa para saber el tipo. ¿Entonces desde cuando se el tipo de una variable en C? desde el momento que escribo el programa. Pero un lenguaje como APL. Si yo escribo a ← 7, a está obteniendo el valor 7 pero además el tipo entero, y si luego digo a ← mensaje, a está obteniendo el valor mensaje y el tipo cadena de caracteres. Entonces en APL el tipo de la variable no se conoce hasta el momento en que se usa. Entonces se dice que el binding es estático o dinámico.

Binding estático: una propiedad es estática cuando no puede variar durante la ejecución de un programa. Otra definición de estático es, cuando se conoce en tiempo de compilación. Pero los lenguajes interpretados también tienen tipos estáticos, entonces no tiene que ver tanto con la compilación. Entonces es mejor decir, que si con el texto del programa se sabe el tipo, entonces el tipo es estático.

Binding dinámico: una propiedad es dinámica cuando varía durante la ejecución o puede variar.

Ejemplo:

Binding de variables. Vamos a ver 4 propiedades. Tipo, alcance, valor, y almacenamiento. Y vamos a ver cuando son estáticas y cuando dinámicas.

	TIPO	ALCANCE	VALOR	ALMACENAMIENTO
ESTATICO	La mayoría de los lenguajes.	La mayoría de los lenguajes.	Casi nunca.	Fortran, COBOL.
DINAMICO	APL, J, Smalltalk y algunos otros.	Basic, APL, Lisp y algunos pocos más.	La mayoría de los lenguajes.	La mayoría de los lenguajes.

La mejor forma de saber si el tipo de variables es estático, es mirar una asignación. Si se tiene: a = b, donde el tipo de a es distinto del tipo de b, esto percibe sin lugar a duda sin ninguna ambigüedad si el lenguaje tiene tipos estáticos o tiene tipos dinámicos. Si para hacer esta asignación se cambia el tipo de a, para que se vuelva del tipo de b, entonces digo que el lenguaje tiene tipos dinámicos. O sea, que si el tipo se copia, se tiene tipos dinámicos. Si el tipo de b, se convierte al tipo de a, y luego se copia, o sea que el tipo se adapta, entonces se tienen tipos estáticos.

Almacenamiento, es la noción que dice cuando la variable tiene lugar físico en memoria donde reside. O dicho de otra manera en algunos lenguajes la variable es una fantasía porque existe la variable pero cuando uno se pregunta dónde está, no está en ningún lado porque no está en memoria. Entonces si el almacenamiento es estático significa que la variable está siempre en memoria, pero si el almacenamiento es dinámico, significa que la variable a veces está en memoria y a veces no. El caso más sencillo de este tipo es el caso de las variables locales de C o de Pascal, son variables que no están en memoria mientras no se ejecute la unidad que las contiene. Esto trae una consecuencia, en 2 ejecuciones distintas de una función las variables pueden ubicarse en distintos lugares.

Alcance de una variable es el conjunto de sentencias que la pueden usar, que ven la variable.

En Basic por ejemplo:

```
10  A = 5
20  Input b
30  if B < 0 then goto 50
40  C = 3
50  X = A + C
```

Si B es menor a 0, el programa da error, porque en Basic no se pueden usar las variables si no tienen valor. Entonces solo se sabe en tiempo de ejecución si la sentencia 50 puede usar la variable C, porque a veces si la puede usar y a veces no.

En variables el valor no tiene casi importancia, las variables son variables y su valor varia.

Clasificación de los lenguajes:

La primera gran clasificación de los lenguajes dice que tenemos (clasificación anterior a la noción de objetos):

Lenguajes estáticos: son los que tienen tipos, alcance y almacenamiento estáticos. Fortran y COBON son conocidos como lenguajes estáticos y **no recursivos**. La recursividad es una noción de que una unidad (una función, un método, un procedimiento) se puede llamar a sí mismo. En ese llamado a si mismo se supone que arranca con todas las variables en su estado inicial, no en el estado que tenia la unidad anterior, o sea, cuando se llama recursivamente a una unidad, la llamada comienza con todas las variables frescas en cero o con sus valores iniciales. Pero además supone que cuando vuelva recuperó los valores anteriores, entonces, se debe tener diferentes lugares donde estén las variables de la primera unidad, los de la segunda y así, o sea se debe tener varias copias de las variables y no se puede hacer esto si las lugares están en un lugar fijo, por lo que el **almacenamiento dinámico es una condición necesaria muy fuerte para que pueda un lenguaje ser recursivo**. No debe interpretarse que todos los lenguajes con almacenamiento dinámico son recursivos pero si **todos los lenguajes estáticos son no recursivos**.

Lenguajes tipo Algol: son los que tienen almacenamiento dinámico, tipos y alcance estáticos, se los nombra de esta manera porque Algol fue quien introdujo la idea del almacenamiento dinámico. (La mayoría C, Pascal).

Lenguajes dinámicos: son los que tienen almacenamiento dinámico y tipos o alcance dinámico. (APL, J, Basic).

Modelo de ejecución

Las dos soluciones clásicas al modelo de ejecución son la interpretada, y la compilada.

En la **solución interpretada** el programa escrito es llevado de la mano por otro programa llamado intérprete que lo va entendiendo y lo va ejecutando. Es más flexible y en consecuencia es más lento.

En la **solución compilada** el programa escrito, se traduce completamente al lenguaje de máquina, es decir no queda un solo residuo del programa original. No quedan las variables, no quedan nombres. Es más rígido y es mucho más eficiente, más rápido.

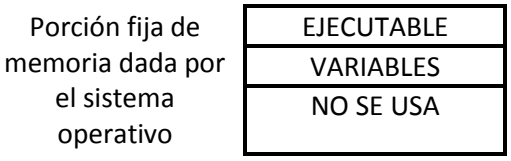
Dependiendo de la estructura semántica del lenguaje, a veces no se puede compilar, o sea, se pueden compilar los programas escritos en lenguaje estáticos y casi sin excepción se pueden compilar los programas escritos en lenguaje tipo Algol, pero solo se puede compilar parcialmente o no compilar en absoluto y tener que interpretar los programas en lenguaje dinámico.

Cuando alguien quiere ejecutar un programa, no está libre de decidir para un lenguaje cualquiera, que hacer, si un compilador o un intérprete.

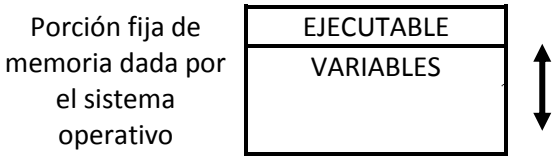
Una visión del modelo de ejecución muy importante es la que se logra viendo como se usa la memoria durante la ejecución. En un lenguaje estático, es más fácil porque las variables están siempre en el mismo lugar. Pero en un lenguaje tipo algol o en un lenguaje dinámico las variables cambian de lugar, aparecen y se borran, entonces se hace más complicada la estrategia de administración de memoria. Pero además el sistema operativo decide si se va a trabajar en el contexto de memorias fija de tamaño fijo, en un contexto de segmento, o en un contexto de memoria virtual.

Supongamos de contexto un sistema operativo donde hay un tamaño fijo, o sea que el programa tiene a su disposición un tamaño fijo. Entonces la vista que se tiene de la memoria es siempre la misma. Ahora hay que ver que hace el lenguaje con ese tamaño fijo que le dio el sistema operativo.

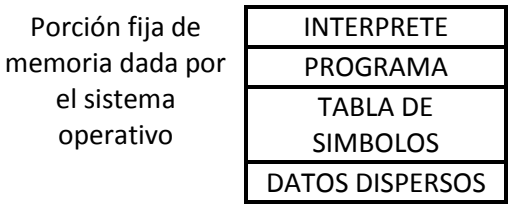
En este contexto, **un lenguaje estático**: se carga el programa en memoria, va a ocupar un cierto tamaño, va a haber una parte que va a ser la parte ejecutable, van a estar las variables y va a haber una parte que no se usa.



En este contexto, **un lenguaje tipo Algol**: va a haber una parte que va a ser la parte ejecutable también, y van a estar las variables, pero estas variables a veces son mas y a veces son menos. Porque si está ejecutando una función cuando termina las variables ya no están, pero si esta ejecución invoca a otra aparecen más variables, y si se llama recursivamente aparecen más variables. Entonces la ejecución avanza o disminuye ocupando más o menos memoria.



En este contexto, **un lenguaje dinámico**: primero tenemos un intérprete, un programa, una tabla de símbolos con los nombres de las variables y la ubicación. Y luego datos dispersos en memoria referenciados por la tabla de símbolo.



Administración de memoria

Uso de la memoria en lenguajes tipo Algol

La primera noción que ocurre muy fuerte en los lenguajes tipo Algol es la existencia de unidades. Las unidades son, métodos, funciones, procedimientos, conjuntos de instrucciones que tienen variables locales. Pero algunos lenguajes admiten **unidades anónimas**, conjunto de instrucciones con dos propiedades, **delimitadas** (con begin-end o {-}) y con **variables locales**. Las unidades con nombre se las puede llamar pero las unidades anónimas, solo se ejecutan si el programa llega hasta ahí. La consecuencia es que las unidades con nombre pueden ser recursivas y las unidades sin nombre no. Ejemplos:

En C el bloque encerrado entre { } es un bloque anónimo:

```
.....
{
    .....
    .....
    .....
}
.....
```

En C, las funciones son bloques con nombre.

```
void funcion() {
    .....
    .....
    .....
}
```

En Pascal, los procedimientos son bloques con nombre.

```
Procedure procedim()
var…
Begin
    .....
    .....
    .....
End;
```

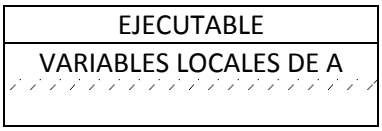
Pero en Pascal no existen los bloques anónimos ya que, a pesar de existir el par de instrucciones begin / end, dentro de ellos no se permiten declarar variables.

```
Begin
    .....
    .....
    .....
End;
```

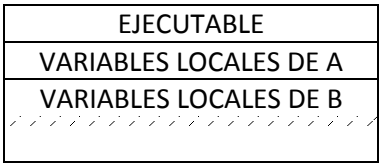
La ejecución de un programa en un lenguaje tipo Algol prácticamente siempre involucra la llamada de un main si es que existe a funciones, etc. Entonces imaginemos una secuencia de llamadas entre unidades.

A→B→C→ C→C→D

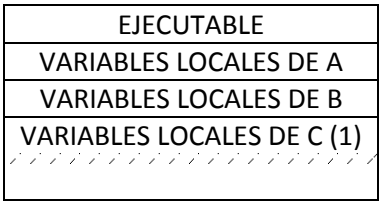
En memoria va a haber:



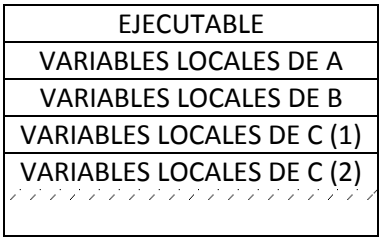
Cuando A llama a B se carga en memoria las variables locales de B:



Cuando B llama a C se carga en memoria las variables locales de C pongamos (1) para no confundir:



Cuando C (1) se llama recursivamente se carga en memoria las variables locales de C (2):



Cuando C (2) se llama recursivamente se carga en memoria las variables locales de C (3):

EJECUTABLE
VARIABLES LOCALES DE A
VARIABLES LOCALES DE B
VARIABLES LOCALES DE C (1)
VARIABLES LOCALES DE C (2)
VARIABLES LOCALES DE C (3)

Cuando C (3) llama a D se carga en memoria las variables locales de D pongamos (3) para saber que fue llamado por C (3):

EJECUTABLE
VARIABLES LOCALES DE A
VARIABLES LOCALES DE B
VARIABLES LOCALES DE C (1)
VARIABLES LOCALES DE C (2)
VARIABLES LOCALES DE C (3)
VARIABLES LOCALES DE D (3)

O sea que a medida que las unidades comienzan a ejecutar se crean espacios para las variables.

Supongamos ahora que termina D, C (3).

EJECUTABLE
VARIABLES LOCALES DE A
VARIABLES LOCALES DE B
VARIABLES LOCALES DE C (1)
VARIABLES LOCALES DE C (2)

y que a C (2) se le ocurre llamar a D. entonces

EJECUTABLE
VARIABLES LOCALES DE A
VARIABLES LOCALES DE B
VARIABLES LOCALES DE C (1)
VARIABLES LOCALES DE C (2)
VARIABLES LOCALES DE D (2)

Entonces esa línea sinuosa significa que el límite va variando subiendo y bajando de acuerdo a que unidad se llame o no se llame.

¿Donde están las variables globales? Generalmente se encuentran después del ejecutable.

EJECUTABLE
VARIABLES GLOBALES
VARIABLES LOCALES DE A
…

¿Se puede quedar sin memoria? Si, puede llegar un momento en el que no entra mas nada.

Entonces ahora podemos comparar las ventajas e inconvenientes de los lenguajes estáticos respectos de los lenguajes de tipo Algol. Los lenguajes de tipo Algol, permiten recursividad, usan menos espacio que los otros lenguajes porque los otros tienen todas sus variables en memoria, los lenguajes tipo algol tienen solo algunas. si por ejemplo el programa del ejemplo anterior tuviese las unidades de la A a la Z, solo están las variables de la unidad A, B, C, y D, el resto de las variables no están. En cambio en los lenguajes estáticos, están todas las variables en memoria aunque no se las use.

Los lenguajes estáticos son no recursivos, pero jamás dejan de ejecutarse porque no les alcanza la memoria. Si no les alcanza la memoria, no les alcanzó nunca la memoria, y una vez que le alcanza le alcanza siempre.

Los lenguajes tipo Algol pueden correr y en un momento dar overflow.

Registro de activación

Al conjunto de variables vamos a llamarlo **registro de activación**. Pero además agrego un dato, en el registro de activación se encuentran las variables más otras cosas. Entonces, los **registros de activación** son una porción de memoria reservada para los datos que manipula una unidad o bloque en ejecución. Cuando en el programa se llama a una función, inmediatamente se reserva en memora el espacio donde contendrá las variables locales de la función, los parámetros de entrada y de salida y todos aquellos elementos que se necesiten para su ejecución. Cuando esta función concluye su ejecución, se procede a liberar la memoria que ocupa el registro de activación y se retorna al bloque llamador. Cada unidad en ejecución tiene un registro de activación.

Ahora vamos a dar vuelta el dibujo para que se parezca a una pila, y vamos a cambiar los lugares donde dice variables por registros de activación, para obtener la **pila de registros de activación**.

REGISTRO DE ACTIVACIÓN DE D (3)
REGISTRO DE ACTIVACIÓN DE C (3)
REGISTRO DE ACTIVACIÓN DE C (2)
REGISTRO DE ACTIVACIÓN DE C (1)
REGISTRO DE ACTIVACIÓN DE B
REGISTRO DE ACTIVACIÓN DE A

Todos los lenguajes tipo algol agregan un elemento a la ejecución de los programas. O sea independientemente de que todas las variables tienen almacenamiento dinámico el lenguaje se reserva para sí mismo una variable estática interna que maneja solo el lenguaje. O sea una variable propia del lenguaje que el programador no sabe que existe se crea en forma estática en un lugar fijo. Tan fijo es eso que se suele asignar de forma estática en un registro de la computadora.

Supongamos maquina con 8 registros y que usa el R8 para esto. La decisión de que registro usar depende de las propiedades de los registros.

El lenguaje a este registro le asigna una función, **siempre está apuntando a la base del registro de activación en ejecución**.

Ahora que ocurre, cuando se hace una compilación de una instrucción para un lenguaje de tipo Algol en lugar de hacer una asociación variable-dirección como los lenguajes estáticos, se hace una asociación donde a cada variable se le asocia la distancia del comienzo del registro de activación al lugar asignado a la variable.

Es decir en un lenguaje de tipo Algol las variables no tienen dirección que cambia. Tienen almacenamiento dinámico, pero dentro del registro de activación tienen un lugar fijo. O que las variables tienen dirección interna fija, no dirección real fija. La dirección **real se compone por comienzo del registro de activación + lugar interno**. Entonces veamos la diferencia entre un lenguaje estático y un lenguaje tipo Algol para el siguiente fragmento de código.

```
Int X, Y, Z;
    .....
    .....
X := Y + Z;
```

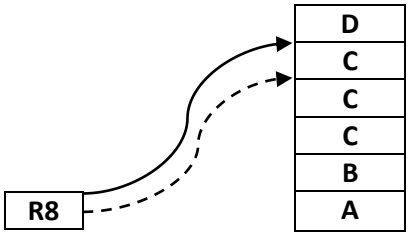
<table>
<tr><th>LENGUAJE ESTATICO (Fortran)</th><th>LENGUAJE TIPO ALGOL (C)</th></tr>
<tr><td>Asociación directa entre variable y dirección de memoria hecha por el compilador, ya no existen más variables, se han convertido en direcciones de memoria</td><td>Asociación variable y desplazamiento dentro del registro de activación.</td></tr>
<tr><td><table><tr><td>X</td><td>400B</td></tr><tr><td>Y</td><td>2AF0</td></tr><tr><td>Z</td><td>310E</td></tr></table></td><td><table><tr><td>X</td><td>10</td></tr><tr><td>Y</td><td>14</td></tr><tr><td>Z</td><td>30</td></tr></table></td></tr>
<tr><td>MOV R1, 2AF0
ADD R1, 310E
MOV 400B, R1</td><td>MOV R1, [R8+10]
ADD R1, [R8+14]
MOV [R8+30], R1</td></tr>
</table>

O sea que se deja de ver las variables como direcciones para verlas con direccionamiento indexado o direccionamiento relativo. Se ven como una base más un desplazamiento. No importa dónde está el registro de activación, lo que importa es que las variables están bien ordenadas respecto de su comienzo.

Ahora ¿Como se hace para que R8 apunte siempre al comienzo del registro de activación actual? El registro R8 se modifica automáticamente por el lenguaje en dos oportunidades. Cuando se crea una nueva unidad, se lo sube inmediatamente al comienzo de la nueva unidad y cundo finaliza la unidad actual, se lo baja, al comienzo de la unidad anterior.

Vamos a suponer que se hace un el llamado de C→D

Inicialmente el R8 apunta a la base de C y cuando se crea D, el R8 debe pasa a apuntar a la base de la nueva unidad, esto se hace de la siguiente manera:



Cuando se crea un registro de activación se ejecuta la instrucción:

ADD R8, (Tamaño del registro de activación de C)

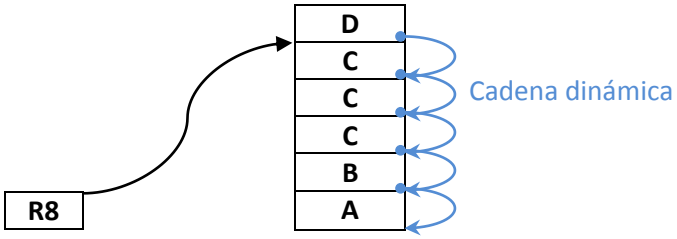
Cuando se suma al R8 el tamaño de C, hay un problema, no se sabe donde es la base de C, para cuando termine de ejecutar D. Entonces lo que se hace es, en un lugar de D, se guarda la dirección de comienzo de C. O sea en uno de los

lugares internos del registro de activación se guarda la dirección de comienzo del anterior. Entonces la instrucción antes de hacer la suma, debe guardar la posición actual del R8.

Vamos a suponer que el lugar donde se guarda la dirección de comienzo del anterior, es el lugar 4 dentro del registro de activación. Entonces:

```
MOV R1, R8                              //guardamos la dirección de comienzo del RA anterior.
ADD R8, (Tamaño del registro de activación de C)  //hacemos apuntar al R8, a la base del nuevo RA.
MOV [R8+4], R1                          //guardamos la dirección de comienzo del RA de C en la
                                        //posición 4 de D.
```

Esto no se hace solo con el llamado C→D, sino que se fue haciendo con cada registro de activación que se apiló.



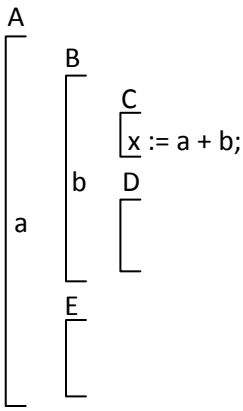
Entonces con estas instrucciones se crea un nuevo registro de activación que sabe donde comienza, el registro de activación de la unidad anterior.

Cuando se destruye un registro de activación la siguiente instrucción hace apuntar al R8, al comienzo del RA de la unidad anterior.

MOV R8, [R8+4]

Se había hablado de que el registro de activación contenía variables y otras cosas, la primer otra cosa que vemos que contiene, es el puntero de la cadena dinámica. Pero además hay variables pasadas como parámetros en la llamada de una unidad a otra.

Este aspecto de los modelos de ejecución de los lenguajes tipo algol es incompleto, porque solo sirve para variables locales. Entonces vamos a hacer un pasito más en relación a las unidades, y recodamos que:



Unidades anidadas: A contiene a B.

Unidades disjuntas: B disjunto de E.

En el contexto de unidades anidadas, aparece la regla de alcance o la forma de definir los alcances o la forma de estimar los alcances es decir. En el dibujo anterior, la asignación usa las variables de las unidades que lo contienen.

Este asunto de poder usar variables de las unidades que la contienen implica que debe estar reflejado de alguna manera el la pila de registros de activación. O sea en tiempo de ejecución si se tiene la situación planteada anteriormente B tiene que saber donde están las variables de A. Entonces en la pila de registros de activación hay otro puntero que describe la estructura léxica o estática o de anidamiento del programa. Eso es lo que permite manejar las variables globales.

Vamos a ver el "cuentito" que se conoce hace tiempo que dice:

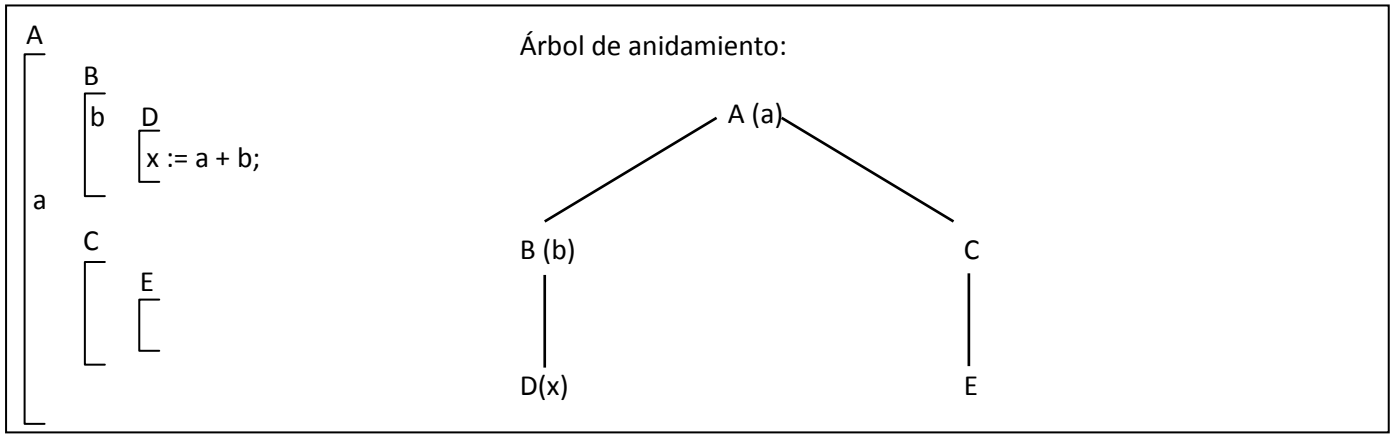
"Si una variable no existe en la unidad local, se la busca en la unidad que la contiene y si tampoco está, se sigue con las unidades más exteriores".

Esto es falso, y lo que tiene de falso, es el tiempo verbal. La frase sería:

"Si una variable no existe en la unidad local, se la busco en la unidad que la contiene y si tampoco está, se siguió con las unidades más exteriores".

¿Por qué en pasado? Porque esto lo hizo el compilador, no hay búsqueda de variables en tiempo de ejecución. Las variables se buscaron en tiempo de compilación y se ubicaron en qué lugar está en el momento que se compiló el programa.

Pongamos mas detalles al ejemplo para ver qué es lo que ocurre. Durante el proceso de compilación el compilador construye lo que llamamos **árbol de anidamiento.**



Si recordamos, el compilador hace una tabla de (variable – dirección dentro del registro de activación). Pero además hace otra cosa, el compilador mira donde está la instrucción. Y luego se fija la distancia desde la instrucción a las variables que utiliza.

Variable	Dirección dentro del RA	Distancia en el árbol
a	**20**	**2**
b	**30**	**1**
x	**18**	**0 (es local)**

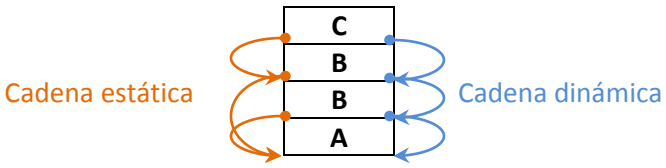
Repasamos, para usar variables globales debo tener un mecanismo nuevo, porque no sé donde están las variables globales, entonces recurado el "cuentito" de primer año. Pero tengo que saber, que la frase es en pasado, porque en tiempo de ejecución no hay búsqueda.

Comentario del profesor: cuando viene un alumno a revisión de final de 3.95, y le pregunto cómo se usan las variables globales y me dice “se busca”. Lo primero que le pregunto es cuál es la clave de búsqueda. Y en tiempo de ejecución no hay claves, no hay nombres.

Entonces la búsqueda se hizo cuando había clave sobre la tabla de símbolos, en tiempo de compilación. **Cada vez que se compila una instrucción se busca en la tabla de símbolos y se deduce en qué lugar está dentro de que registro de activación.**

Entonces como se registra esto, cada registro de activación contiene otro puntero más que apunta al padre en el árbol de anidamiento.

Suponemos la siguiente secuencia de llamados: A→B→B→C



Entonces cada registro de activación tiene dos punteros, uno al que lo llamó cadena dinámica, y el otro a su padre en el árbol de anidamiento cadena estática.

Ahora ¿Como se hace la suma?

Seguir la cadena estática 2 pasos.
MOV R1, [R8+20]
Seguir la cadena estática 1 paso.
ADD R1, [R8+30]
Nada, porque la variable es local.
MOV [R8+18], R1

La suma sigue siendo la misma, el problema es que debe desplazarse. Para usar la variable **a** se debe seguir la cadena estática 2 paso. Luego para usar la variable **b**, debo seguir la cadena estática 1 paso. Y para usar la variable **x**, no se debe hacer nada, porque la variable x es local.

Entonces la ejecución de variables globales involucra ir recorriendo punteros de una nueva cadena llamada cadena estática.

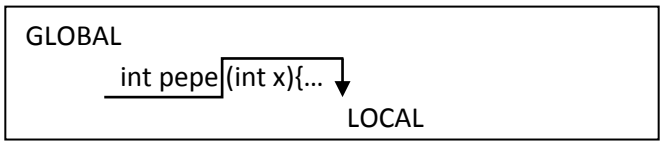
Clase 3. Construcción de la cadena estática

¿Cuál es el alcance del nombre de una unidad?

O sea yo tengo una unidad **Z**, esa unidad tiene un nombre, y me pregunto ¿Z es local a Z o es global?

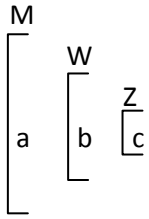
Si fuera una variable seria local a Z, pero como es una unidad, si el nombre de la unidad fuera local a Z no se la podría llamar. Porque no se ve. **Entonces el nombre de una unidad es global a la unidad**.

Ejemplo en C:



El nombre de una unidad se ve de afuera, y el nombre de los parámetros se ven de adentro.

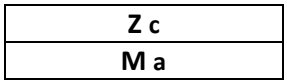
Si tenemos lo siguiente:



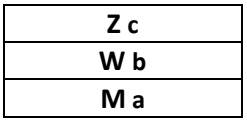
Como Z es global a la unidad misma, es conocida por todas las instrucciones de W.
¿Pero será conocida por todas las unidades de M también?

1- Hipótesis: M puede llamar a Z.
2- Hipótesis: M no puede llamar a Z. entonces la única forma de que M llegue a z es por el camino M→W→Z.

Como quedaría la pila de registros de activación en cada caso.



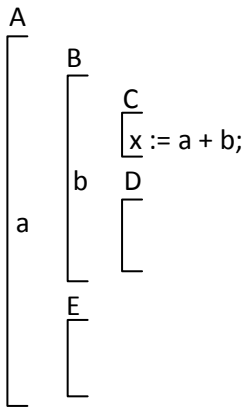
Hipótesis 1



Hipótesis 2

Como falta el registro de activación W, no están las variables locales a W por lo tanto no se puede ejecutar la instrucción.

Entonces tenemos que, el alcance de una unidad, es global a sí misma, y local, a quien la contiene. Esto lleva a la pregunta de quién puede llamar a quien, o sea que llamadas están permitidas y cuáles no, para esto hacemos la siguiente matriz. Donde en vertical se pone la unidad que llama, y en horizontal la unidad llamada.



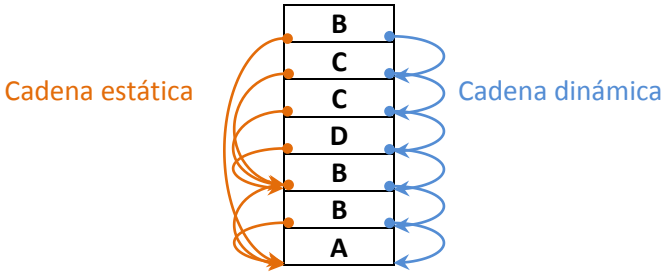
	A	B	C	D	E
A	R	L	X	X	L
B	G_1	R	L	L	G_1
C	G_3	G_2	R	G_1	G_2
D	G_3	G_2	G_1	R	G_2
E	G_2	G_1	X	X	R

X no puede llamar.
R recursiva o también G_1.
L local.
G_1 global de distancia 1.
G_2 global de distancia 2.
G_3 global de distancia 3.

Esta matriz no es armada por el compilador, el compilador hace el árbol de anidamiento y con eso tiene todo. Esta matriz es una forma de explicar que cosas están ocultas y que cosas están a la vista en un conjunto de unidades anidadas pero solo un recurso didáctico.

Ahora sabemos que cada unidad, tiene una cierta distancia en el árbol de anidamiento. Respecto de la unidad que está llamando.

Suponemos la siguiente secuencia de llamados: A→B→B→D→C→C→B



¿Cómo hago para lograr que B apunte a A, en la primera oportunidad?

Cuando la llamada es local el puntero de la cadena estática coincide con el puntero de la cadena dinámica.

Entonces como hacemos la construcción. **"La cadena estática se construye usando la cadena estática"**.

Pero para poder hacer esto tengo que hacer lo siguiente:

1- Crear el puntero de la cadena dinámica.
2- Copiar el puntero de la cadena dinámica en el puntero de la cadena estática. O sea suponemos que la llamada es local.
3- Mirar el árbol de anidamiento y calcular la distancia **n** en el árbol de anidamiento.
4- Seguir **n** veces la cadena estática ya construida.

Vamos a suponer que la cadena dinámica está en 2 y la cadena estática está en el lugar 4 dentro del registro de activación. Para armar la cadena estática utilizo las siguientes instrucciones:

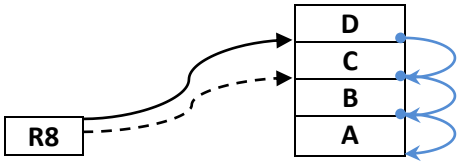
```
MOV [R8+4], [R8+2]
MOV R7, R8 //guardo el puntero al RA en ejecución.
MOV R8, [R8+4]
…                    } n veces
MOV R8, [R8+4]
MOV [R7+4], R8 //ajusto el puntero de la cadena estática.
MOV R8, R7 //restauro el puntero al RA en ejecución.
```

La relación entre la unidad llamada y la unidad llamadora en término de las funciones queda determinada en forma estática. Si la llamada es B→C entonces el vínculo es fijo y estático siempre que no haya objetos involucrados. Si los hay, el vínculo queda resuelto a través de los punteros.

Las llamadas internas dentro del ejecutable están todas resueltas en forma absoluta con direccionamientos fijos a lugares fijos, siempre hablando en lenguajes de tipo algol. Ahora las variables están en lugares que no sé donde están porque van a ir cambiando de acuerdo a la ejecución. La resolución de cómo acceder a las variables es disponiendo de una dirección fija o un registro llamado puntero de pila que en nuestros ejemplos es el R8, que apunta al registro de activación en ejecución y después con todos los líos de la cadena estática y la cadena dinámica.

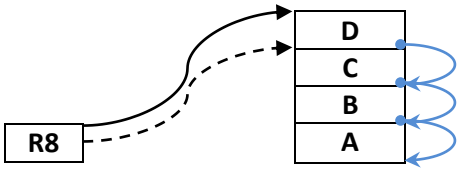
Entonces cadena estática y cadena dinámica, refiere solo a variables. No a las partes ejecutables de las unidades. Las partes ejecutables de las unidades están en lugares permanentemente fijos.

Hasta ahora vimos un modelo en el cual el registro de pila (R8) apunta siempre a la base del registro de activación en ejecución. En este caso cuando se realiza la siguiente secuencia de llamado: A→B→C→D, en el último llamado, para mover el R8 de la base de C a la base de D, lo que se hace es sumar, el tamaño del RA de C. Entonces se dice que el nuevo registro de activación lo crea la unidad llamadora.



En este modelo las variables se ubican con la suma:
R8 + offset

Pero hay otro modelo en el cual el registro de pila apunta al tope del registro de activación. Con lo cual para la secuencia de llamados A→B→C→D, en el último llamado hay que sumar al R8 el tamaño de D y no el tamaño del registro de activación actual. El único que puede hacer esto, porque conoce su tamaño es la unidad D es decir que el registro de activación lo crea la unidad llamada.



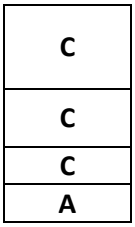
En este modelo las variables se ubican con la resta:
R8 - offset

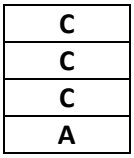
¿Por qué éste último modelo es mejor que el anterior?

El otro es más fácil de entender, pero **el segundo es mejor porque** estas instrucciones están solo una vez por unidad, no importa si una unidad es llamada 100 veces, **la creación de un registro de activación está siempre en el mismo lugar y una sola vez por unidad**. Entonces esto hace al programa un poco más compacto.

El estándar, prácticamente todos los lenguajes, usan la segunda solución.

Vamos a suponer que se tiene la siguiente secuencia de llamado: A→C→C→C, me pregunto si varios registros de activación de la misma unidad en la misma ejecución pueden o no tener tamaños distintos.





La respuesta es, para C, C++, Pascal **no**, para ADA y Algol **sí**.

Entonces si hay lenguajes donde esto no puede pasar, estos lenguajes tienen alguna propiedad que no hemos visto todavía, y eso es lo que se va a tratar a continuación.

Clasificación de variables

Variables estáticas

Los lenguajes que tienen solos variables estáticas se llaman estáticos. Otros lenguajes pueden tener algunas variables estáticas y otras no, en general a esos lenguajes se los llama tipo Algol.

Variables semi-estáticas

Las variables que hemos estado visualizando dentro de los registros de activación se han caracterizado por tener un offset fijo dentro del registro de activación. Si la variable **k** está siempre en el mismo lugar significa que todas las anteriores tienen siempre el mismo tamaño. Entonces si el registro de activación tiene siempre el mismo tamaño, todas sus variables tienen siempre el mismo tamaño. A estas variables que están dentro del registro de activación pero que conservan siempre el mismo tamaño se las llama semi-estáticas. Las variables semi-estáticas no tienen por qué estar siempre en el mismo lugar, pero se tiene la certeza de que su tamaño es fijo.

Variables semi-dinámicas

Las variables semi-dinámicas son aquellas que en distintos registros de activación, pueden tener distintos tamaños. Pero una vez que se ubican en el registro de activación se quedan quietas, no cambian de tamaño. Cuando se crea el registro de activación, no puede cambiar su tamaño porque tiene otro registro de activación atrás. Son variables que en distintos registros de activación pueden tener distintos tamaños, pero empecemos a pensar, que ¿Pueden ser enteros más largos y enteros más cortos? N**o**. ¿Son reales más largos? **No**. ¿Son estructuras más largas? **No**. ¿Son mecanismos más complicados? **No**. Entonces ¿Qué son? que no cambian de tipo porque los lenguajes de tipo algol tienen tipos estáticos pero tienen tamaño diferente. Y la respuesta es que **son arreglos**. Las variables semi-dinámicas son arreglos. ¿Pero que una unidad tiene 2 índices y otra tiene 3 índices? **No**. Si no cambia la cantidad de subíndices, entonces ¿Qué cambia? **Los limites**. Estas variables tienen límites variables. Entonces **las variables semi-dinámicas son arreglos con límite variable**. Es cuando escribimos en C: int x[k]; donde k es otra variable. Entonces el tamaño del límite es una variable.

Por oposición aquellas variables en las que solo se puede poner un número en la definición de un arreglo: int x[k]; donde k es un numero, son variables semi-estáticas.

Primera condición de las variables semi-dinámicas, cuando se crea el registro de activación hay que conocer el tamaño de las variables, ya el registro de activación no es como antes de un tamaño fijo, entonces necesito saber el tamaño de las variables que pueden cambiar de tamaño. Entonces para crear el registro de activación que contiene la instrucción: int x[k]; donde k es variable, necesito conocer k con anticipación. La consecuencia entonces es que k debe ser una variable global a la unidad que contiene la declaración. No puede ser local, porque si es local no existe la variable todavía, y si no existe la variable todavía no tiene valor. Tiene que tener valor antes de empezar la ejecución, por lo tanto tiene que estar en otro registro de activación.

¿Que nos pasó? Con esta estrategia, se acaba de morir, la teoría de que cada variable está en una posición fija dentro del registro de activación ya no existe más. Entonces los lenguajes que tienen variables semi-dinámicas tienen que tener algo nuevo que no se ha visto todavía.

Para entender esto vamos a ver como se guardan los arreglos en memoria y como se usan los arreglos. Supongamos:

var z [4..40, 6..8] of integer;

La memoria es lineal, no cuadrada, está la dirección 45, luego la 46 y luego la 46. Entonces la única forma de guardar arreglos de varias dimensiones es desarmar el rectángulo y ponerlo en una línea. Por lo tanto este arreglo se va a guardar de la siguiente manera:

	6	7	8
4			
5			
...			
40			

→ desenrollando la matriz por filas

Z_{4-6}	Z_{4-7}	Z_{4-8}	Z_{5-6}	Z_{5-7}	Z_{5-8}	...	Z_{40-8}

→ desenrollando la matriz por columnas

Z_{4-6}	Z_{5-6}	Z_{6-6}	Z_{7-6}	Z_{8-6}	Z_{9-6}	...	Z_{40-8}

Ahora los 4 números de las dimensiones del arreglo (4, 40, 6 y 8) ¿Existen en algún lugar en tiempo de ejecución? ¿Necesito los límites en tiempo de ejecución? La pregunta está incompleta porque me falta saber si el lenguaje verifica límites. Hay lenguajes que con esta declaración si se hace: z[2][7] te deja usarlo, y otros lenguajes que no dejan porque verifican que el índice está fuera del límite.

Entonces si el programa ejecutable te dice, que el índice está fuera del límite, tiene que conocer el límite. ¿Cómo sabe sino? Entonces, si el lenguaje verifica limites, los 4 índices están presentes en memoria con if de algún tipo. Si estoy usando los índices i, j en algún lugar del programa va a haber:

```
if ( i < 4 ) error;
if ( i > 40 ) error;
if ( j < 6 ) error;
if ( j > 8) error;
```

Entonces va a haber in if, por cada límite por cada uso de las variables para verificar que no se salga del límite. Por eso muchos lenguajes no verifican límites. El programa se vuelve más lento.

Entonces concluimos en que si el lenguaje verifica limites, (4, 40, 6 y 8) van a estar en memoria en instrucciones. Si el lenguaje no verifica limites, parecería que no se necesita, pero en realidad si se necesitan porque todavía nos falta algo.

Si quiero ver donde está z[i][j] para cualquier i, j. decimos que la dirección para cualquier elemento del arreglo decimos que es:

La dirección del primero Z_{4-6} + un cierto desplazamiento, donde el desplazamiento es (i - 4)*(8 – 6 + 1)+(j - 6)*T

Si i vale 4 significa que estoy en la primera fila no hay ninguna fila antes. Pero si i vale 5 hay una fila entera antes de llegar a la fila 5. Entonces (i – 4) me dice cuantas filas hay antes de la fila que busco. Y (8 – 6 + 1) es el tamaño de la fila. Entonces el producto, me dice cuantas variables hay antes de la que busco en términos de fila pero hay que sumarle el desplazamiento dentro de cada fila (j - 6). Todo esto es la cantidad de variables que hay dentro de la que busco. Pero a esto todavía lo tengo que multiplicar por el tamaño T de cada elemento (2 si son enteros, 4 si son reales, por 8 si son doublé, etc.).

Esta cuenta se debe hacer en el programa ejecutable, cada vez que aparezca z[i][j]. O sea que si se tiene la instrucción z[i][j] := z[i+1][j+1]. Hay que calcular 2 veces las cuentas.

Entonces concluimos que también se necesitan los límites. Excepto el 40, esto es porque es por fila, por caluma es lo mismo pero no necesito el 8. Entonces si se tiene un arreglo de 3 dimensiones, en memoria para hacer las cuentas necesito 5 de los 6 límites.

Pero qué pasa si la variable no fue declarada así, sino que fue declarada:

var q [m..n, r..s] of integer;

Las cuentas son exactamente las mismas solo que cambiando los límites de la siguiente manera:

La dirección del primero Z_{m-r} + un cierto desplazamiento, donde el desplazamiento es (i - m)*(s – r + 1)+(j - r)*T

Las cuentas son siempre necesarias. Entonces los arreglos en memoria requieren los límites para verificar límites, y casi todos los limites para saber dónde cae cada componente del arreglo.

Veamos qué pasa cuando los limites son variables y cuando los limites son constantes:

Entonces un arreglo está siempre compuesto por límites y datos. Si es semi-estático va a tener siempre el mismo tamaño, si es semi-dinámico, va a tener límites y datos, pero en otra ejecución quizá tenga otros límites y datos.

semi-estáticos	Limites: 4, 40, 6, 8	Datos
semi-dinámicos	Limites: 4, 20, 6, 9	Datos
semi-dinámicos	Limites: 4, 60, 6, 11	Datos
semi-dinámicos	Limites: 6, 12, 7, 21	Datos

Cuando es semi-dinámico, la diferencia a cuando es semi-estático es que el tamaño va a ir cambiando. Pero hay otra cosa más que va cambiando acá, en los arreglos semi-estáticos coy a tener (4, 40, 6 y 8) siempre, porque son constantes.

En las variables semi-dinámicas voy a tener, (4, 20, 6 y 9) en una ejecución, en otra (4, 60, 6 y 11) y en otra (6, 12, 7 y 21). Voy a tener distintos valores que dependen de las variables m, n, r y s de cuando se crean cada registro de activación.

Entonces en las variables semi-estáticas tengo todas constantes. En las variables semi-dinámicas tengo los valores que tenían las variables m, n, r y s cuando se creó el registro de activación. Entonces hay que tener cuidado, porque el límite inferior no es el valor de m, es el valor que tenia m, cuando se creó la variable, después puede haber cambiado de valor. Pero la variable quedó fija en ese tamaño. Entonces no me basta con guardar m, n, r y s tengo que guardar los valores que tenían m, n, r y s cuando se creó la variable.

En las variables estáticas los límites son fijos, entonces no tengo porque guardar una copia en cada registro de activación si son siempre el mismo número, entonces a estos se los manda al comienzo del ejecutable, donde están las instrucciones de if, o donde están las instrucciones que hacen las cuentas. No se las pone en el registro de activación.

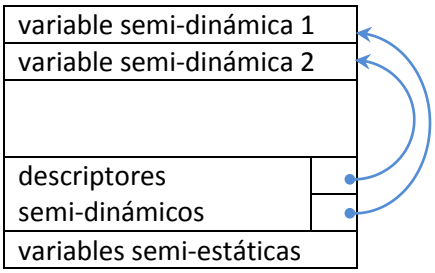
Pero si las variables son semi-dinámicas tengo que tener los datos y los límites del arreglo.

Entonces ya tenemos una composición, dentro de un lugar están solo los datos de las variables semi-estáticas pero están los datos y los limites de las variables semi-dinámicas.

Desde el punto de vista de la implementación lo primero que se dice es un trabalenguas: “los limites de las variables semi-dinámicas son ellos mismos semi-estáticos”. ¿Qué quiere decir? Que su tamaño es fijo.

Entonces los que se hace ahora es desacoplar los límites de los datos se ponen los límites por un lado los datos por otro lado y se agrega un puntero que dice: estos son los límites y los datos están acá.

A esto se lo suele llamar descriptores de las variables semi-dinámicas. Entonces ahora dentro del registro de activación se pone primero todas las variables semi-estáticas. Entonces para todas estas que no cambian su tamaño, vale el modelo que dice que las variables están en el comienzo del registro de activación más un desplazamiento. Y luego para las semi-dinámicas lo que se hace es poner los descriptores de las variables. Entonces como los descriptores tienen tamaño fijo, cada descriptor sé en qué lugar está y nunca se va a correr, y luego se pone, la variable semi-dinámica 1, y se hace que el puntero apunte. Se pone la variable semi-dinámica 2 y se hace que el puntero apunte y así sucesivamente. Se van poniendo las distintas variables semi-dinámicas y se van cargando los punteros. Entonces no importa que las variables semi-dinámicas cambien de tamaño porque con base mas desplazamiento yo llego al descriptor y el descriptor me dice donde está el dato. Entonces las variables semi-dinámicas son más costosas de utilizar.



Fíjense que bien que habíamos empezado, las variables estáticas están en una dirección fija pero no tenemos recursividad, hacemos las variables semi-estáticas en una pila y las ubicamos por base mas desplazamiento, anda bastante bien, bastante rápido pero después aparecen las variables globales entonces hay que hacer mas cuentas. Después aparecen los arreglos tenemos que hacer cuentas, luego los arreglo semi-dinámicos entonces además de hacer cuentas tenemos que dividirlos por un lado poner los limites y por otro lado poner los arreglos. Y para usar la semi-dinámica, base mas desplazamiento llego al comienzo, base más un cierto byte tengo el otro puntero que m lleva al dato. Y obviamente ahí tengo todos los límites para hacer todas las cuentas que necesito.

Entonces las variables semi-dinámicas son más costosas de usar pero son más flexibles. Y como vamos a seguir viendo, flexibilidad se paga siempre con tiempo de ejecución.

Variables dinámicas

Las definimos con una propiedad más fuerte que las variables semi-dinámicas. Entonces decimos que las variables dinámicas pueden cambiar de tamaño en cualquier instrucción. Las variables semi-dinámicas adquirían su tamaño al comienzo de una unidad y luego se quedaban quietas las variables dinámicas pueden en cualquier instrucción cambiar de tamaño. Entonces son más complicadas que las variables semi-dinámicas, tienen mayor flexibilidad en el manejo de su tamaño.

Las variables dinámicas pueden ser:

Con nombre: han casi desaparecido, se usaron muy pocas veces, un ejemplo es el caso de Algol, donde se las llamó arreglos flexibles. Ejemplo si definimos:

flex int a [1 : 0]; //la forma de declarar arreglos flexibles en algol es poniendo limites sin sentido.

La consecuencia que trae esto es que yo puedo escribir en cualquier lugar:

a := (4, 1, 7, 8, -5, 3);

Y a partir de este momento, a es como se hubiese sido definido entre 1 y 6: flex int a [1 : 6];

Pero en la instrucción siguiente puedo escribir:

a := (0, 0, 3, 2); // a es como se hubiese sido definido entre 1 y 4: flex int a [1 : 4];

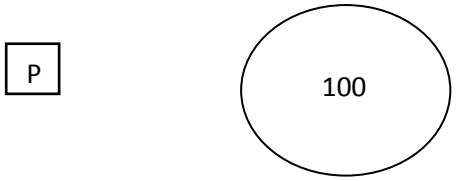
O sea que en cualquier instrucción le puedo cambiar el tamaño, y si b fuese un arreglo flexible y digo:

b := a; //b adquiere el tamaño de a.

Nada me impide decir a[3] := 22; //donde estoy asignándole 22 a un componente particular.

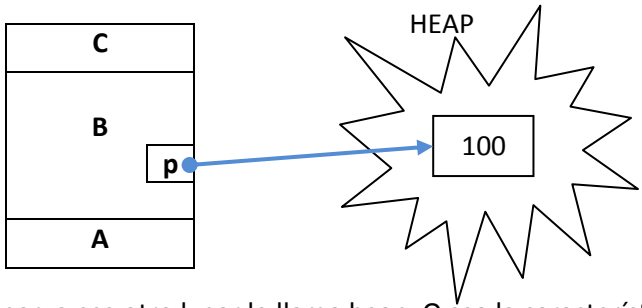
Anónimas: son mucho más comunes, son las clásicas que se ven en C, C++ y muchos otros lenguajes a partir de una función malloc, o new o create. Que crea una variable anónima. Ejemplo:

p = (int*) malloc (100); //en esta instrucción hay 2 variables: la variable p, y la variable 100 que no tiene nombre.

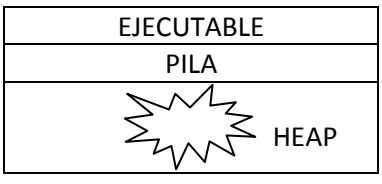


p está en la pila, los 100 bytes no tengo idea.

Entonces como se guardan las variables dinámicas anónimas: voy a tener un registro de activación, dentro de ese registro de activación puedo tener los componentes que tengo u otros, pero voy a tener una variable que va a ser un puntero. Y cuando ejecute una instrucción como la mencionada anteriormente, en otro lugar que no describimos sino que le damos nombre de **heap**, se crea un bloque de 100 bytes apuntado por p, pasa lo mismo con new o con otras palabras en otros lenguajes.



Entonces se va a crear un bloque que está en otro lugar, a ese otro lugar lo llamo heap. O sea la característica esencial de las variables dinámicas anónimas es que están fuera de la pila.



Fuera de la pila en otro lugar diferente de memoria hay otras variables que fueron creadas con malloc o con new, están relacionadas a través de punteros, obviamente si tengo una estructura recursiva o algo más complicado puedo tener punteros entre ellos, por ejemplo, arboles, listas, colas.

Vamos a ver qué pasa con las variables dinámicas con nombre.

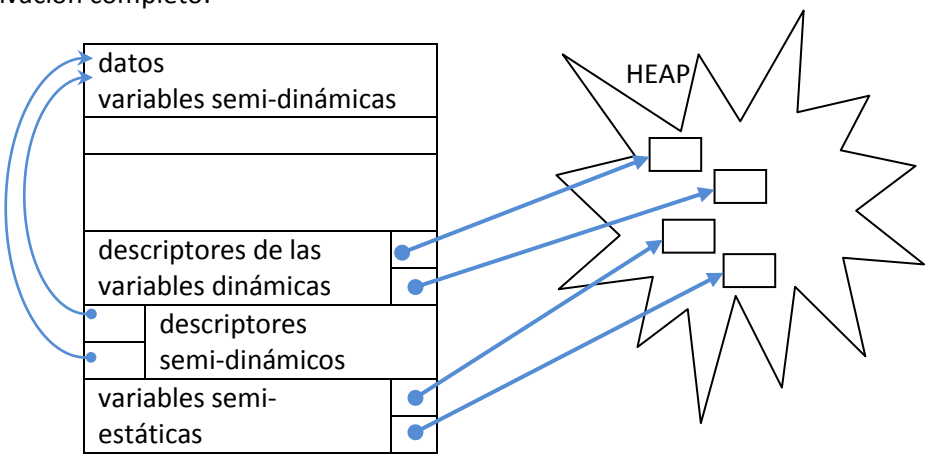
Las variables dinámicas con nombre, tienen esta característica pero no cambian de tipo. Son siempre arreglos de una dimensión o arreglo de dos dimensiones no les aparecen nuevos índices ni le desaparecen índices, o sea si la declaré con dos índices, tiene siempre dos índices. El tamaño de la matriz puede cambiar en cualquier lado pero siempre conserva las dimensiones. Entonces qué pasa, estos tienen la misma estructura que las variables semi-dinámicas, es decir estas variables dinámicas con nombre, tienen una parte que es un descriptor, un puntero y los datos separados.

Las variables dinámicas anónimas tienen exactamente lo mismo que las semi-dinámicas, entonces ¿Cuál es la diferencia? Que los descriptores cambian en cualquier momento, entonces no pueden estar en la pila.

Entonces la parte de datos de las variables dinámicas con nombre, también están en el heap, también están fuera de la pila.

O sea que las variables dinámicas, todas, se fueron parcialmente al heap. En el caso de que sean anónimas todas las variables se fue al heap, en el caso de que sea con nombre el descriptor queda en la pila que es de tamaño fijo y los datos se fueron al heap.

Dibujamos entonces un registro de activación completo:



Entonces las variables semi-dinámicas están completamente en la pila pero divididas en dos mitades descriptores por un lado y datos por otro, las variables semi-estáticas están todas juntas, las variables dinámicas con nombre tienen punteros que van afuera de la pila hacia el heap. Y las variables dinámicas anónimas, ¿Se las visualiza con qué? Algunas de las variables semi-estáticas son punteros y esos punteros apuntan también al heap. Entonces las variables dinámicas con nombre tienen descriptores al heap, las variables dinámicas anónimas se basan en un puntero que apunta también al heap.

Este es el modelo completo de registros de activación, a las variables semi-estáticas se accede como base del RA mas desplazamiento, a las semi-dinámicas, base mas desplazamiento mas puntero, y a las dinámicas base mas desplazamiento mas puntero, a las anónimas base mas desplazamiento y puntero.

Ahora no todos los lenguajes tienen esto, algunos lenguajes tienen una cosa y otros tienen otra, por ejemplo, C y Pascal, tiene variables semi-estáticas y a través de punteros dinámicas también. ADA tiene semi-estáticas, semi-dinámicas y dinámicas también. Algol tiene semi-estáticas, semi-dinámicas dinámicas anónimas y dinámicas con nombre, este lenguaje se caracterizó por tener todos los tipos de variables.

Variables		
	Estáticas (están siempre en el mismo lugar). Tienen tipo fijo y tamaño fijo.	**Lenguaje estático**
	Semi-estáticas (por estar dentro de un registro de activación su ubicación física cambia). Tienen tipo fijo y tamaño fijo.	**Lenguaje tipo Algol**
	Semi-dinámicas (son arreglos con límites variables). Tienen tipo fijo y tamaño variable cada vez que se crea el registro de activación.	
	Dinámicas (anónimas ó con nombre). Tienen tipo fijo y cambian de tamaño en cualquier momento.	
	Dinámicas con tipo dinámico.	**Lenguaje dinámico**

La presencia de variables que cambian de tipo en cualquier momento, fuerzan la estrategia de construcción de la tabla de símbolos y ya no mas una pila de registros de activación las variables ya se empiezan a administrar a través de una tabla de símbolos.

Ejemplos de 3 lenguajes dinámicos típicos:

Basic: el alcance es dinámico porque una variable solo se puede usar si tiene valor. Ahora si yo miro la celda de memoria donde supuestamente está almacenada la variable miro 2, 4, 6 bytes que corresponden a la variable, no hay forma de darse cuenta si ese valor es basura que quedo del anterior en memoria, o si es el correcto. Entonces la única forma de saber si una variable tiene valor es tener una lista de variables con valor asignado. La una manera de saber si una variable se puede usar es que se encuentre en esa lista. Esta lista es el principio de la tabla de símbolos, después se le agregan algunos atributos y ya está.

APL: es igual a Basic, pero hay variables locales. En Basic original solo hay variables globales.

J: es igual a APL, pero se pueden crear grupos de variables globales. En cualquier punto del programa uno puede decir abandono el grupo de variables del “cliente” y empiezo a usar el grupo de variables del “stock”. Pero se sigue cumpliendo que una variable solo se puede usar si tiene valor. Y esto me condena a tener una tabla de símbolos.

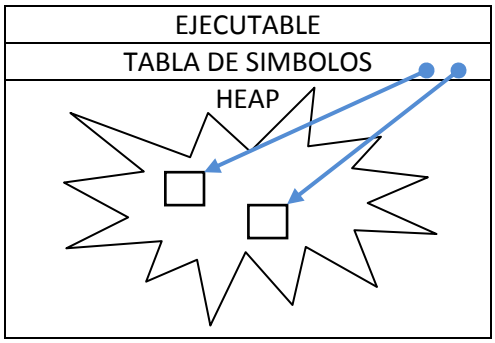
¿Que contiene una tabla de símbolos?

Nombre	Ubicación	Tipo	…

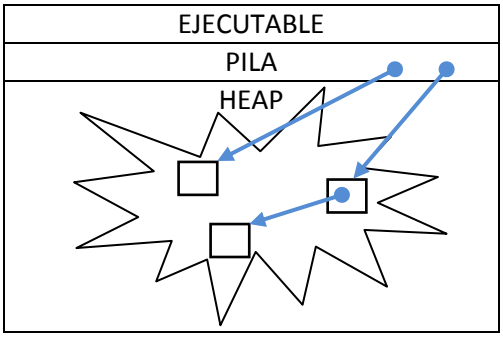
En tiempo de ejecución está el nombre de la variable en memoria.

En cada instrucción yo necesito saber qué tipo tienen las variables cuando voy a hacer la suma a + b. Necesito saber el tipo de a y de b, justo en este momento, no el que tenían hace 2 instrucciones atrás, ni el que van a tener instrucciones mas adelantes. Entonces necesito una especificación de tipo dentro de la tabla de símbolos.

Una pequeña comparación entre los lenguajes tipo algol y los lenguajes dinámicos desde el punto de vista de la administración de memoria.



TIPOS DINAMICOS



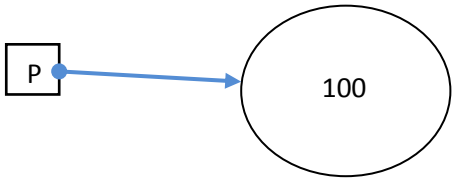
TIPO ALGOL

En los lenguajes tipo algol el heap puede tener punteros que apuntan a otro lado del heap mismo.

Entonces la visión que se tiene del heap es distinta, para los lenguajes dinámicos, el heap es algo totalmente visible a través de la tabla de símbolos, para los lenguajes tipo algol, uno va a una variable que es un puntero que me lleva a algo que está en el heap.

Ahora esta condición que tienen los lenguajes tipo algol de tener punteros dentro del heap permite que ocurra el siguiente fenómeno.

Puedo tener un puntero p, que apunta a un tamaño de 100 bytes. Es anónimo y está en el heap. Lo más importante es que el usuario puede modificar el puntero.



Si decimos p = q; si copiamos el puntero, ese p, se va a apuntar a otro lado y ya no ve los 100 bytes que tenía antes.

Pero si yo escribo *p = 3.6; estoy alterando una parte del bloque de 100 bytes.

Lo importante de esto es que si yo uso "*" no estoy modificando el puntero, o sea que sigue apuntando al bloque, estoy modificando el bloque en sí mismo.

Entonces yo puedo escribir instrucciones que alteren el puntero o que alteren el dato apuntado.

Si hacemos p = q; creo una situación que se llama creación de **garbage**.

La creación de garbage, es tener algo en el heap inaccesible porque se ha perdido el puntero que me permitía acceder.

Pero si yo hago free(p); hago desaparecer el bloque de 100 bytes y entonces el puntero se queda apuntando a algo que no existe más. En este caso se suele hablar de **dangling reference** o **punteros colgados**.

Estos son los 2 males de los punteros explícitos. Garbage y punteros colgados.

Esto pasa solo con variables dinámicas anónimas, con las variables dinámicas con nombre no pasa este mecanismo porque el globo redondo tiene nombre.

Solo pasa cuando tengo variables dinámicas anónimas, porque tengo un puntero que administra el programador y un bloque de datos que administra el programador.

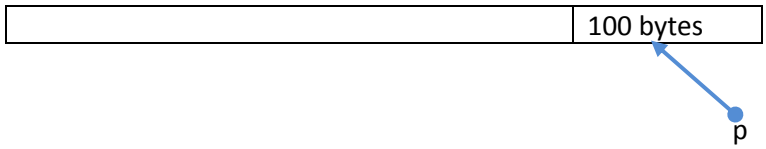
Esto no puede pasar con variables dinámicas con tipos dinámicos, porque siempre la tabla de símbolos los está apuntando. Entonces en los lenguajes dinámicos jamás hay un bloquecito que no se pueda usar. Y jamás hay un puntero que apunte a cualquier lado.

En los lenguajes tipo Algol pueden aparecer punteros que apunten a algo que no existe y puede haber bloques de memoria inaccesibles.

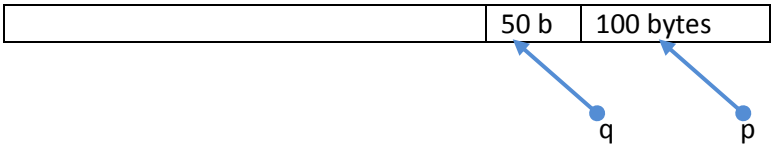
Hay un lio clásico que es el siguiente, cuando empiezo a hacer manipulaciones con variables dinámicas ya sea con tipo variable o con tipo fijo, el lio clásico es que yo las estoy moviendo de lugar. Por ejemplo:

```
p = malloc(100);
q = malloc(50);
free(p);
p = malloc(200);
```

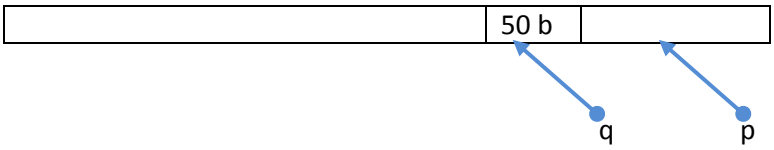
Que es lo que pasa con estas 4 instrucciones, pasa que al final de la memoria se crea un bloque de 100 bytes apuntado por p.



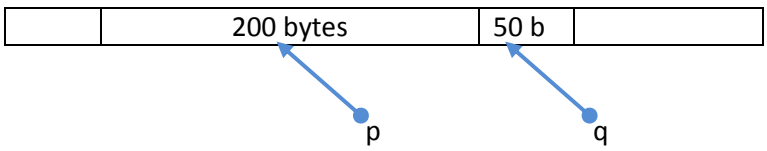
Inmediatamente después se crea un bloque de 50 bytes apuntado por q.



Con la tercera instrucción se elimina el bloque apuntado por p.



Pero con la cuarta se crea un nuevo bloque de 200 bytes apuntado por p.



No hice ningún lio, está bien el puntero p, está bien el puntero q y están bien los bloques de 200 y 50 bytes, pero fabriqué un agujero.

Entonces el uso de variables dinamices anónimas o con nombre en los lenguajes tipo algol van fabricando huecos entre las variables. Las variables ya no están todas pegadas como en la pila.

En lenguajes dinámicos por ejemplo si asigno a una variable un tipo más grande y ya no entra entonces se corre más adelante.

Entonces puede pasar que por mal uso de la memoria, por haber fabricado muchos huecos, no me alcance la memoria. La memoria dejó de ser compacta para estar muy fragmentada.

Entonces esto implica una cierta necesidad de administrar el heap.

El heap en lenguajes dinámicos

¿Cuál es la principal tarea que tengo con el heap en lenguajes dinámicos? Sacar todos los agujeros, es decir compactar. Es decir lo mismo que pasa cuando uno desfragmenta un disco.

¿Entonces si yo tengo una tabla de símbolos cual es el algoritmo de desfragmentación? Recorro las ubicaciones de la tabla de símbolo, y me fijo cual está más cerca del fondo y si no está inmediatamente en el fondo la corro al fondo. Y así sucesivamente. Se recorre la variable más cerca del fondo y se la corre lo más atrás posible.

El defecto del lenguaje es que en el heap hay 2 problemas, numero uno garbage bloques inaccesibles y numero dos huecos o sea memoria fragmentada. Los lenguajes dinámicos solo tienen fragmentación. Porque todos los bloques usados están apuntados por la tabla de símbolos. Lo que se ve en la tabla de símbolos es lo único que sirve. Pero en los lenguajes tipo Algol tengo fragmentación y garbage.

Al procedimiento de recorrer la tabla de símbolos y apilar hacia el fondo las variables eliminando los huecos, se lo llama garbage colector. Pero no son verdaderamente garbage colector, son solamente compactadores.

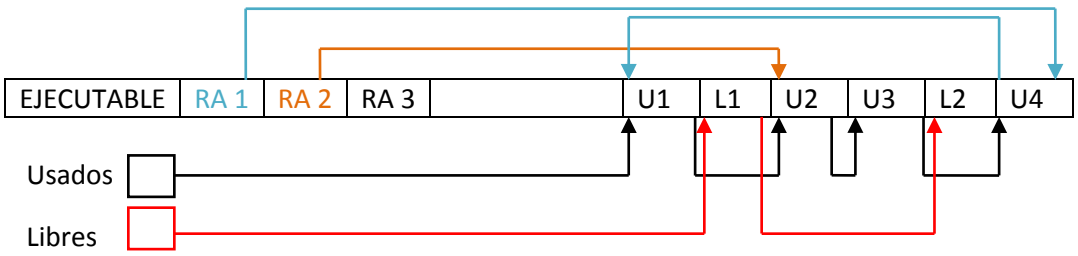
El heap en lenguajes tipo algol

En los lenguajes tipo Algol el lio es mucho más grande porque hay bloques que se perdieron y si se perdieron como hago para recuperarlos. Vamos a ver la estrategia del heap para ubicar los bloques perdidos.

Los punteros colgados no son problema del heap. Solo es capaz eventualmente de administrar el garbage.

Para resolver el problema de la eliminación de garbage y huecos en lenguaje tipo algol debemos hacer lo siguiente:

Vamos a suponer la siguiente estructura lineal:



Los bloques libres son consecuencias de una variable que se borró.

La primera cosa es que nunca hay 2 lugares libres juntos, si los hubiese hay que formar uno solo mas grande.

El lenguaje mantiene 2 listas que el programador no sabe que existe. Una de las listas tiene una cabecera de usados que apunta al primer usado ese primer usado apunta al segundo usado y así sucesivamente hasta el último. Lo mismo pasa con una lista de libres.

Cuando un usuario pide 100 bytes de memoria, no se usan solo 100, se usan 102, 104 108, para guardar datos de administración de los bloques.

¿Qué pasa con malloc? Cuando se usa, se busca con algún criterio (best fit, worst fit, o fisrt fit) en la cadena de libres y ahí se lo pasa a la cadena de usados. Si no hay ningún libre que me alcance, se va al principio de la memoria y se inserta en la lista de usados el espacio correspondiente.

¿El free que significa? Tomar el bloque declararlo libre y mira si al lado hay otro libre para juntar. Es decir el malloc particiona bloques libres y el free eventualmente reúne bloques libres.

Entonces básicamente malloc y free es pasar bloques de una lista a la otra.

En el ejemplo que se puso no hay ningún registro de activación que apunte a U3, entonces eso es garbage. Entonces **garbage** son bloques que están en la lista de usados, pero que el usuario no lo puede usar porque no tiene punteros que lleguen a él.

Entonces cual es el problema, que el registro de activación empieza a avanzar, el heap empieza a avanzar y se pueden chocar.

¿Cómo se arregla esto? Tengo que hacer 2 cosas, primero descubrir cuales son garbage y pasarlos de la lista de usados a libres. O sea hace un free implícito de todos los garbage. Y segundo correr hacia el fondo los que quedan.

¿Cómo sería el algoritmo de garbage collection en lenguajes tipo Algol?

1. Recorrer la lista de usados colocando una marca. Para lo cual cuando se fabricó el bloque hubo se hubo que haber reservado un espacio para la marca.
2. Para cada puntero de la pila visitar el bloque apuntado y borrar la marca. Seguir los punteros del heap borrando la marca.
3. Recorrer la lista de usados y declarar libres los que tienen marca.

Un algoritmo de este estilo elimina los bloques basura, pero no compactó, los huecos siguen estando. Motivo por el cual hay que seguir, y hacer algo más.

4. Recorrer la lista de usados, buscando el que está más cerca del final, y si no está justo al final correrlo. Y además actualizar los punteros desde el registro de activación. Hacer todo esto es muy costoso.

Todos los lenguajes dinámicos hacen compactación de su memoria.

Prácticamente no hay ningún lenguaje tipo Algol que haga garbage collection, están los conceptos de cómo hacerlo pero la mayoría no lo hace.

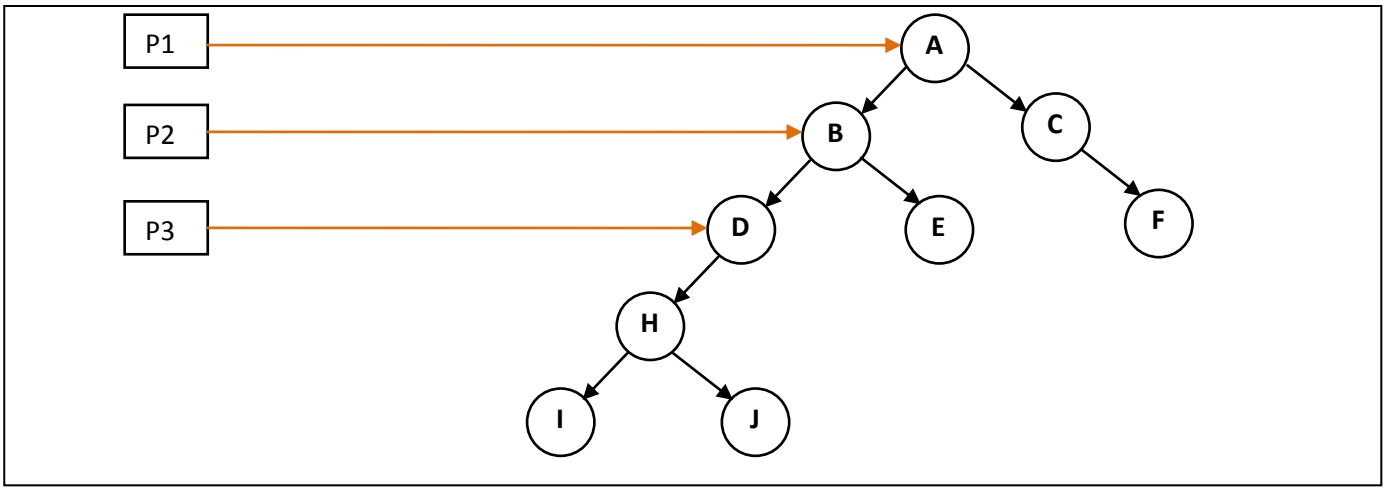
Todavía hay un problema, hay un aspecto no considerado dentro de los lenguajes tipo Algol. Llegamos al garbage collection porque el heap se fue sobre la pila o porque la pila se fue sobre el heap. Es decir llegamos a esta situación porque el programa no tiene suficiente memoria para ejecutarse. E inventamos un algoritmo que recupera parte de esa memoria, eliminando los bloques usados, eventualmente compactando, pero en ese proceso de recuperar bloques usados tengo que recorrer los arboles o grafos que creó el usuario, y para recorrer esas estructuras necesito una pila. Pero donde voy a crear esa pila si no tengo memoria.

En realidad hay una condición más acá, este recorrido de los punteros de pila se haga sin recursividad y el recorrido de los punteros de usuario se tiene que hacer sin consumo adicional de memoria. Eso significa que tiene que haber un recorrido de arboles sin pila.

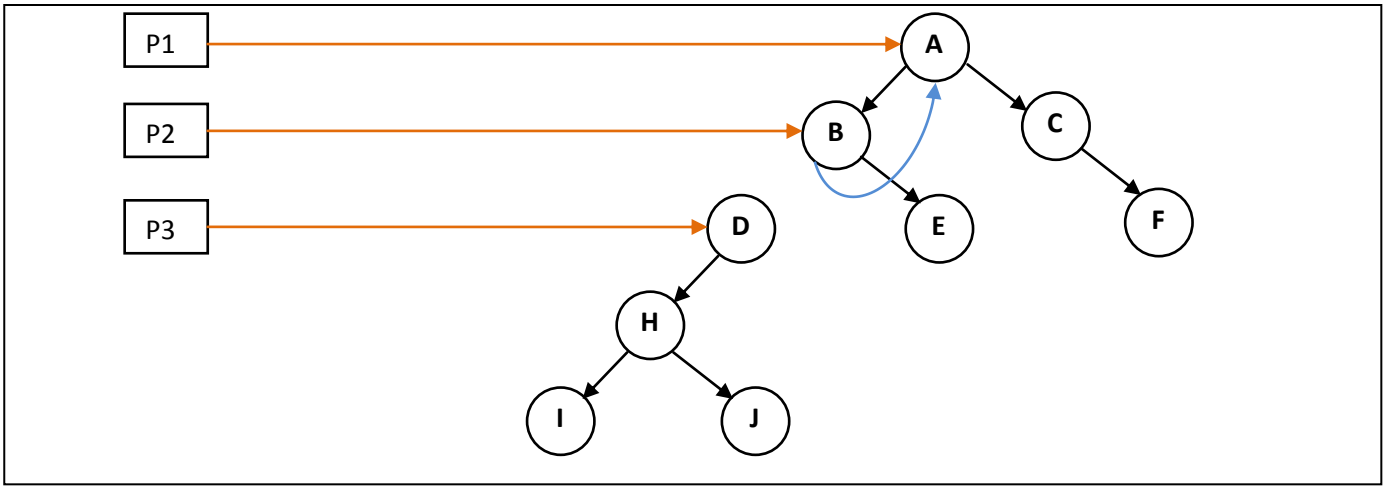
Veamos como seria el recorrido.

Hay un algoritmo que se basa en 3 punteros, p1, p2 y p3.

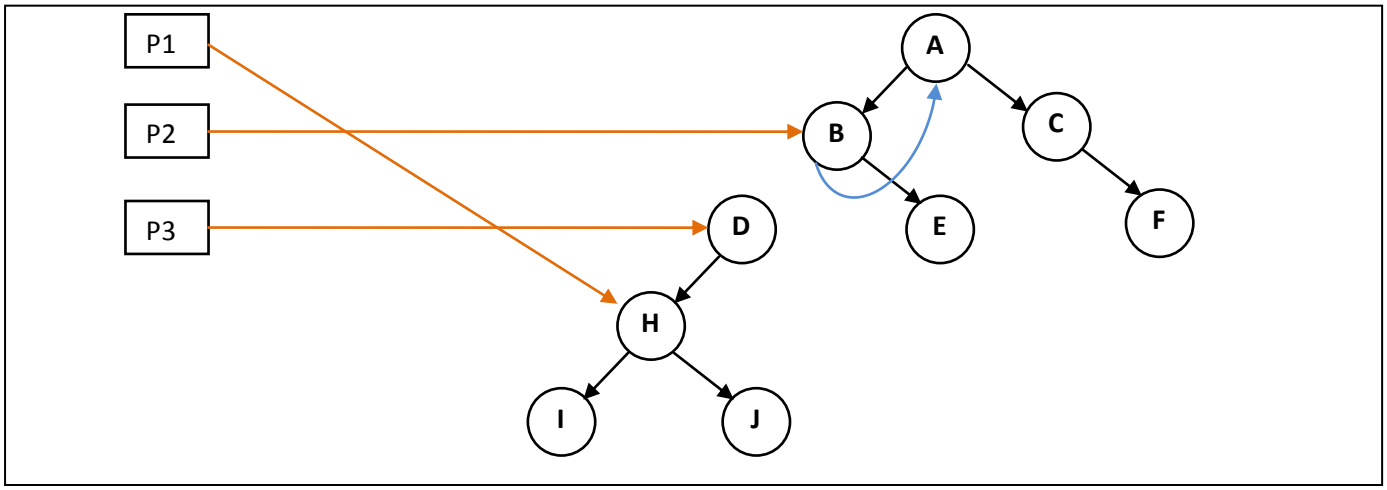
Con el primero de los punteros apunto a la raíz. Con el segundo al hijo izquierdo si es que lo hay, y con el tercero apunto al hijo izquierdo del nodo anterior.



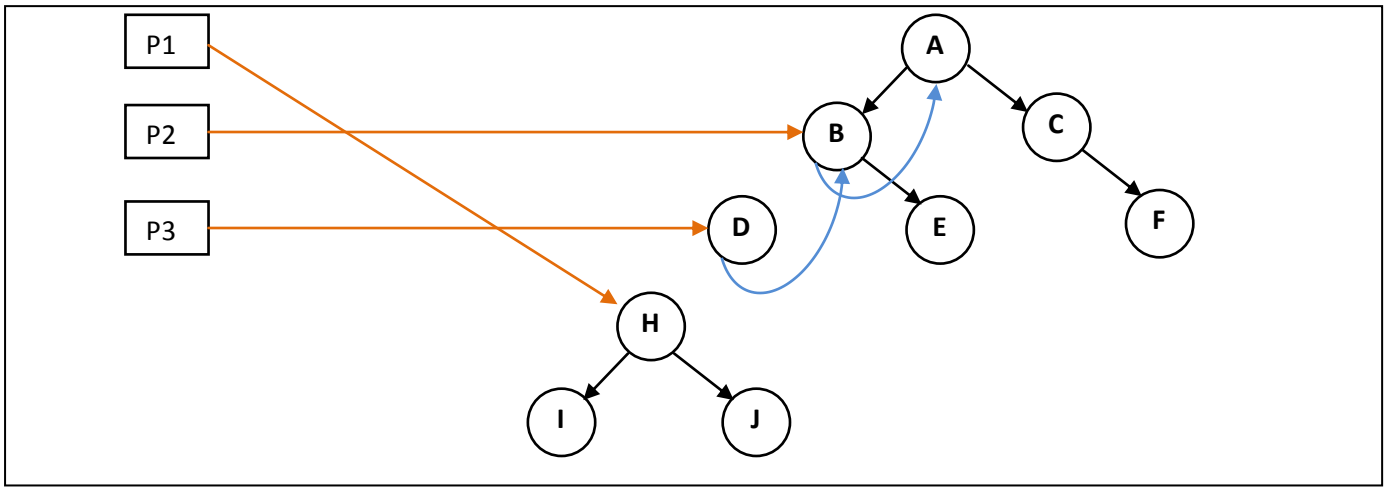
Como quiero seguir para el lado izquierdo y no tengo más punteros, hago que el nodo apuntado por el del medio apunte al padre.



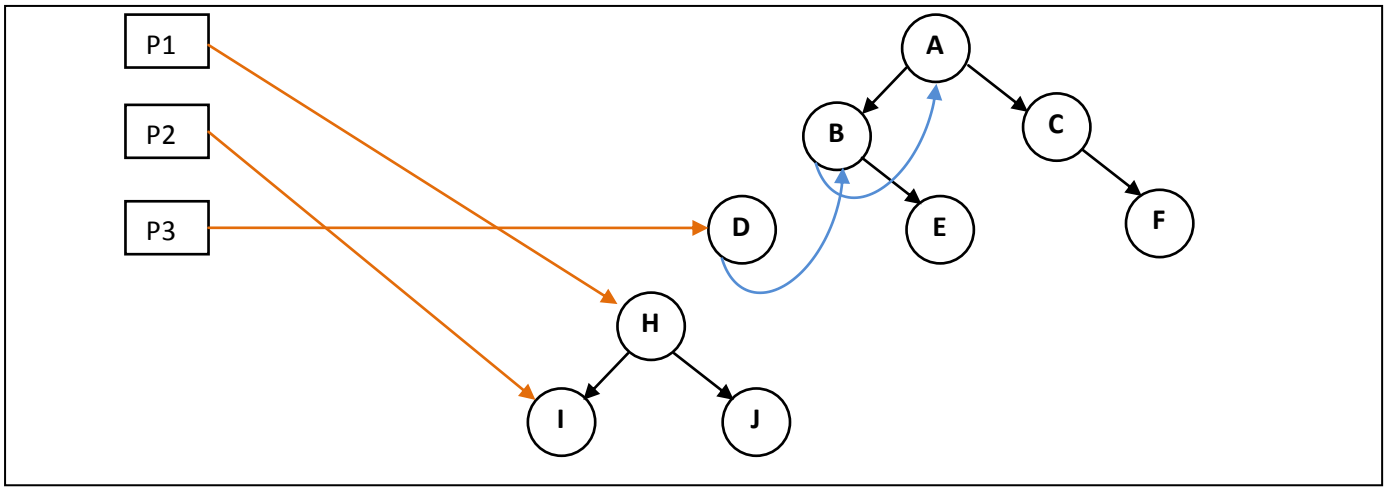
Ahora tengo 2 caminos para llegar al padre, entonces no necesito más el puntero p1 y lo uso para apuntar a H.



Ahora que tengo a alguien apuntando a H, hago apuntar a D al padre.



Ahora tengo 2 formas de llegar a B, por el puntero 2, y por el puntero izquierdo de D entonces al puntero 2 no lo necesito más y lo hago apuntar a I.

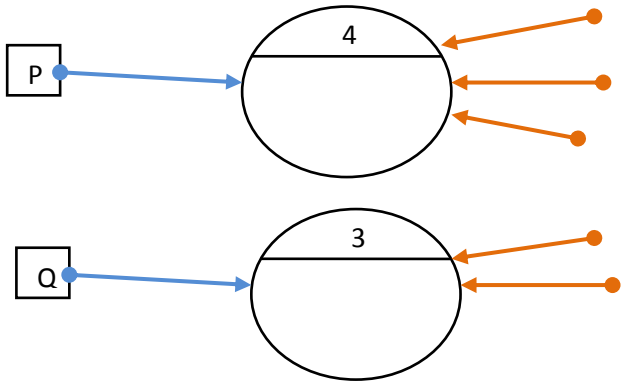


Esto lo puedo hacer infinitas veces. Acá no estoy apilado, ni llamando recursivamente. Estoy solo dando vuelta punteros. A esta técnica se la conoce, como técnica de **inversión de punteros**.

Cuando llego a la hoja hago lo que tengo que hacer con la hoja y restauro el puntero. Entonces puedo recorrer el árbol hacia abajo y hacia arriba. Sin pila y sin recursividad, es decir sin uso de memoria de más de 3 variables.

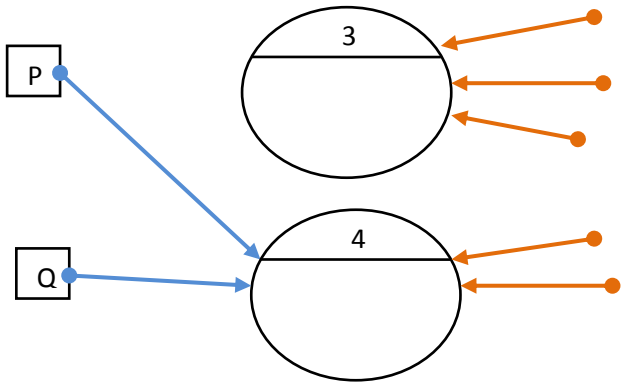
A esta estrategia todavía le falta algo, porque cuando se produce la necesidad de garbage collection el programa se queda seco, porque está haciendo garbage collection. El programa se para y no ejecuta una instrucción mas hasta que no termine el garbage collection. Entonces esta estrategia aunque es correcta, tiene el defecto que está **concentrado temporalmente**. Entonces para evitar ese defecto hay un segundo algoritmo no tan bueno, pero que no está concentrado en el tiempo y es el algoritmo de **contador de referencia**.

El algoritmo de contador de referencia se basa en la siguiente idea. Yo tengo un puntero que apunta a un bloque en el momento de su creación se fabrica un contador que dice: hay un puntero que apunta acá. Ejemplo:



El algoritmo de garbage collection por contadores de referencia se basa esencialmente en la copia de punteros. Entonces en la instrucción:

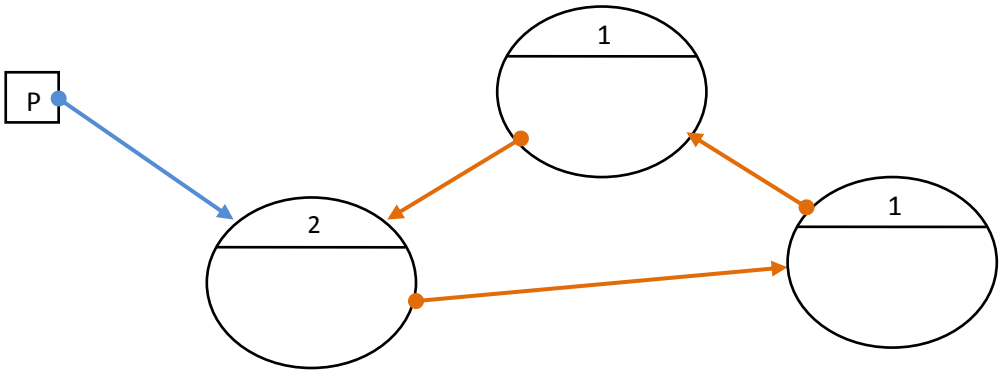
P = Q; al contador de referencia de P se le resta uno y al de Q se le suma uno.



Entonces cuando un contador de referencia llega a cero, se ha producido garbage y ya se puede eliminar.

Este algoritmo tiene la enorme ventaja de que no se fabrica un proceso complejo y largo de búsqueda de datos. Sino que en cada instrucción de manejo de punteros se fija si no se ha fabricado garbage.

Pero tiene un problema de recursividad:



Si en estas circunstancias hago P = otra cosa. Quedan todos con el contador de referencia en 1. Y no detecto garbage. O sea que no sirve para estructuras circulares.

Este algoritmo se usa bastante porque es más simple de programar y porque no carga nunca al programa con una actividad pesada en un momento dado de tiempo. Pero tiene un defecto no solucionable dentro del algoritmo.

Entonces que tenemos hasta ahora:

El heap tiene una estructura más complicada que la pila en los lenguajes tipo algol, el heap tiene, fragmentación y bloques inaccesibles.

Hacer garbage collection en lenguajes dinámicos significa solo compactación.

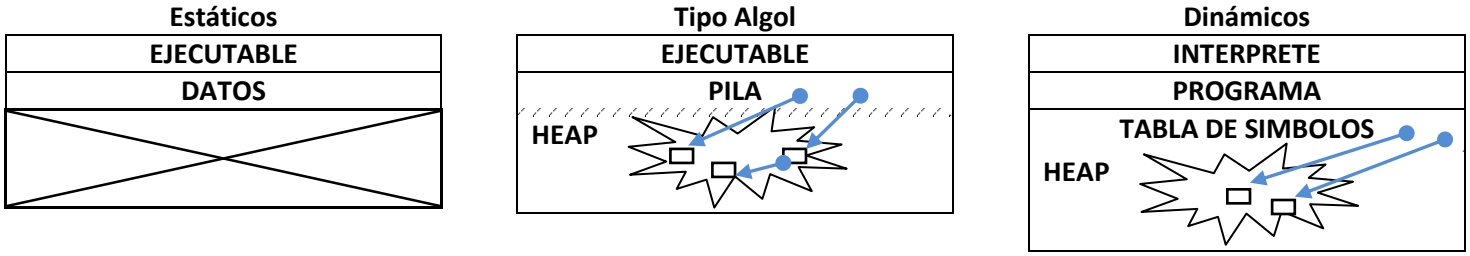
En los lenguajes tipo Algol no se conoce ningún lenguaje comercial que tenga garbage collection. Y el garbage collection significa 2 cosas, se descubren los bloques que no están siendo referenciados y luego realizar la fragmentación.

Por problemas de concentración se suele usar el algoritmo de contador de referencia que no es tan bueno como la técnica de inversión de punteros.

Clase 4 parte 2. Interacciones con el sistema operativo

Hasta ahora hemos estado viendo que pasa con el uso de memoria por parte de los diferentes lenguajes de programación. Y cada vez que hablábamos de la organización de la memoria hacíamos un rectángulo. Y esto pasa porque suponíamos un sistema operativo con una partición fija.

Entonces **si tengo una partición fija** quiere decir que tengo un lugar donde se guardan los programas. Entonces voy a tener el dibujito que ya hemos hecho anteriormente.



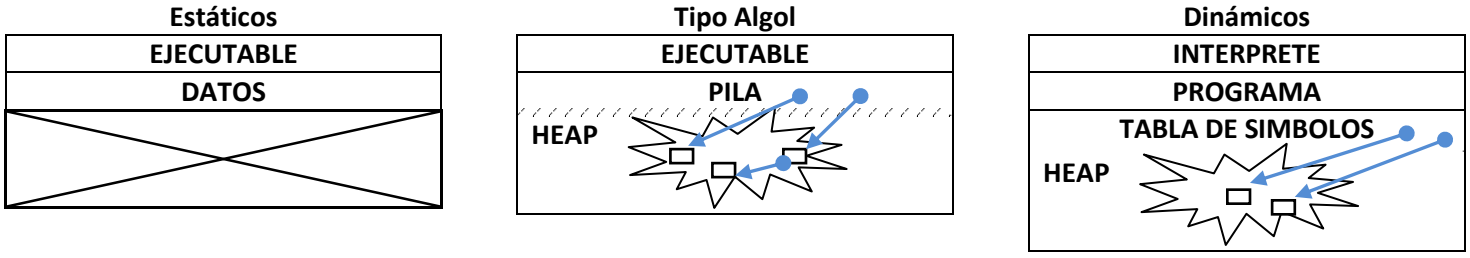
Con lenguajes estáticos el programa entra en memoria o no entra.

Con lenguaje tipo Algol, puede entrar en un momento pero puede que se deje de ejecutar porque la pila y el heap se chocaron.

Con los programas dinámicos pasa lo mismo puede que en un punto del programa no se pueda seguir mas por falta de memoria.

Pero en todos es más o menos así, el programa se puede ejecutar o no se puede ejecutar. Si tengo una sola partición y bueno, es lo que tengo.

Entonces **si tengo varias particiones fijas** la diferencia es que tengo algunos bloques de memoria grande y otros más chicos entonces la decisión de cuando tengo varias particiones donde lo pongo.



Con lenguajes estáticos se conoce el tamaño del programa perfectamente, entonces se lo pone en una partición donde entre. Se elije la partición mas chica donde el programa entre y ya está.

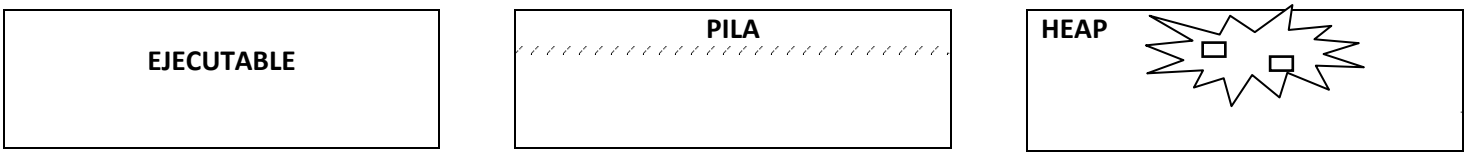
Con lenguaje tipo Algol se complica, porque el mayor consumo de memoria de los lenguajes tipo algol está en la zona de datos más que en el ejecutable pero lo que ve el sistema operativo es el ejecutable, es decir, el sistema operativo ve un archivo que contiene el ejecutable que requiere 200k y no sabe, ni tiene idea de que tamaño va a usar en tiempo de ejecución para las variables. Entonces si al sistema operativo lo dejaran solo, dice que el ejecutable entra y lo metería en la partición más chica. Y apenas comience a ejecutar no hay lugar para la pila. Entonces hay que tomar previsiones de que tamaño debo dejar para los datos hay que estimar. Y esa estimación depende del nivel de recursividad, del tamaño de los arboles que voy a manejar, etc. De una gran cantidad de cosas difíciles de predecir. Entonces el compilador le

tiene que decir al sistema operativo en algún lugar del ejecutable, que tamaño ponerle. Y esto en general, el lenguaje tampoco lo sabe. Porque el lenguaje en general cuando ve los datos declarados no se da cuenta si va a haber mucho nivel de recursividad, si se van a construir grafos muy grandes o muy chicos, no sabe qué espacio darle. Entonces el lenguaje pone un valor por defecto. Y le dan la oportunidad al programador, para que el programador le diga al compilador, la cantidad de memoria que necesita el programa.

En los **segmentos** el problema es parecido a los de varias particiones fijas la diferencia, los segmentos son una serie de particiones de la memoria físicas donde el sistema operativo construye una partición dinámicamente. O sea al momento de ejecutar el programa le asigno una partición y el tamaño de la partición la puedo elegir. O sea la diferencia entre particiones fijas y segmentos, es que segmentos me permite elegir el tamaño de la partición. Y todo lo que dijimos para particiones fijas también vale para segmentos. Donde el programador le dice al compilador que tamaño necesita, el compilador le dice al sistema operativo y éste construye un segmento de ese tamaño.

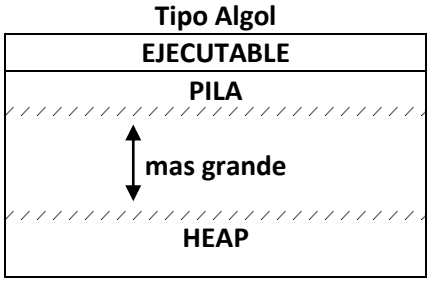
Cuando el sistema operativo puede manejar **varios segmentos** por programa especialmente la línea Intel se caracteriza por tener 4 segmentos, uno para la parte ejecutable, una segunda para la parte de los datos, y para la pila y otra para cualquier cosa que queramos hacer.

Entonces cuando tenemos varios segmentos supongamos que tenemos 3 segmentos. Para los lenguajes de tipo Algol lo que se va a hacer es poner el ejecutable en un segmento, la pila en otro segmento y el heap en un tercer segmento. Este ejecutable como es de un tamaño fijo, se lo va a acomodar, al tamaño que me interesa por lo tanto no hay pérdida de memoria. La pila y el heap, se van a ir construyendo como hemos visto. Salvo que no van a estar en el mismo espacio de direccionamiento, cada uno va a tener su espacio de direccionamiento independiente.



Los varios segmentos ayudan pero poco, porque ya no se produce más el choque de la pila con el heap, pero se produce el choque de la pila con el fondo, o el choque del heap con el fondo.

Finalmente con **memoria virtual** tengo un cuadrado igual que todo lo demás, y su única diferencia es que es mucho más grande. Miles o millones de veces más grande.



La pila y el heap están tan lejos que no me preocupo más, ni hablo de garbage collection. Porque nunca hay conflictos. No ha desaprovechamiento de memoria real y los lenguajes tipo algol funcionan perfectamente. Entonces la solución es abunda la memoria virtual, por lo tanto no hay conflicto. Es una solución tipo fuerza bruta pero es muy eficiente y es obviamente la que se está usando porque desaparece el conflicto de falta de memoria. O de posibles choques de memoria. ¿Pero se está usando mucha memoria? No, porque como solo se asignan páginas reales a lo que se está

usando en este momento tendrá 2 o 3 páginas para el ejecutable, otras 2 o 3 páginas para los bloques del heap que están en uso y listo. Se usa mucho menos memoria real que cualquier otro.

Problemas de concurrencia

Está interacción está vinculada con las hebras de ejecución, estas son una propiedad de la ejecución del programa que quedan a caballo entre propiedades del sistema operativo y propiedades del lenguaje.

Entonces vemos un dibujo de las posibles situaciones que habría desde el punto de vista de la ejecución concurrente:

Vamos a ver 2 ejemplos de lenguaje C y ADA.

C es un lenguaje no concurrente.

ADA es un lenguaje concurrente, que quiere decir, es que 2 procedimientos, 2 funciones, 2 métodos de un programa en ADA se ejecutan simultáneamente.

Entonces hay una típica concurrencia organizada por el sistema operativo, que es la típica concurrencia de Linux que ejecuta programas en C. Ninguno de los programas que se ejecutan bajo Linux, sabe que el otro existe. Se ejecuta cada uno por sí solo, como si tuviera la máquina para él solo.

Pero también puedo ejecutar en un sistema operativo que no soporta concurrencia, un programa en un lenguaje que soporta concurrencia, como es ADA. El sistema operativo ni se da cuenta que hay concurrencia y el lenguaje hace ejecutar sus partes de forma concurrente.

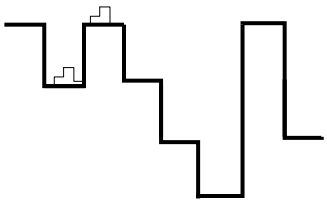
Y finalmente tengo la variación más complicada que es cuando los dos son concurrentes.

Cuando hay concurrencia tiene que haber un task manager que decida a quien le toca la CPU. Si el sistema operativo es concurrente obviamente que ese task manager es parte del sistema operativo. Pero si la concurrencia es de procedimientos de un programa de un lenguaje, es el programa quien administra la concurrencia, entonces tenemos un Process scheduller o un procedure scheduller que está dentro del programa ejecutable. Que lo pone el compilador.

		Sistema Operativo	
		Concurrente	No Concurrente
Lenguaje	Concurrente	Hay una situación un poco más complicada que se va a tratar a continuación. Por ejemplo un programa en ADA bajo Linux.	La concurrencia es administrada por el lenguaje. Es el lenguaje quien decide pedazo de programa se ejecuta ahora.
	No Concurrente	La concurrencia es administrada por el sistema operativo.	Nada para decir

El sistema operativo ve procesos o programas en ejecución entonces tiene:

Programa 1
Programa 2
Programa 3
Programa 4



Se van ejecutando el 1, después va al 2, después al 1, después al 3, etc. Se van ejecutando según su prioridad. Pero cuando el lenguaje además es concurrente. Mientras el tiempo se le asigna a un programa, divide su tiempo entre varios procedimientos o funciones.

Acá ha hebras de ejecución donde tenemos un hilo más grueso que es el programa y luego lo más fino que componen la hebra más gruesa que son los diferentes procedimientos o funciones que están en ejecución.

Pero acá para poder seguir debo responder la siguiente pregunta. ¿Se conocen? Es decir, ¿El lenguaje sabe que está corriendo bajo un sistema operativo concurrente? ¿El sistema operativo sabe que el proceso es concurrente? La respuesta es que hay casos en que si, y hay casos en que no.

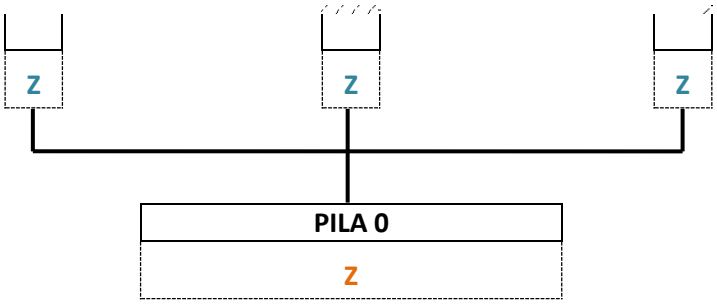
Si hay concurrencia y no se tratan hay 2 administradores de scheduling de CPU. El primero es del sistema operativo, que decide le toca al programa 2 y luego dentro del programa 2 hay un segundo administrador que dice dentro mío le toca a la función tal. No hay forma de dar prioridad que mezcle distintos hilos de ejecución solo se le puede dar una prioridad interna a cada programa, pero no puedo mezclar las del programa 1 con las del programa 2.

Cuando se conocen, hay un administrador de scheduling, se ha establecido un contrato entre el sistema operativo y el lenguaje. En ese contrato la arquitectura clásica es el lenguaje delega al sistema operativo la conmutación entre sus propias hebras de ejecución. Es decir, ahora la conmutación no es del programa 2 al programa 1 sino que es del proceso 5 del programa 2 al proceso 3 del programa 1.

No importa qué modelo sea. Si el lenguaje es concurrente, el se ocupa de la administración de la memoria del programa. ¿Cómo se hace eso?

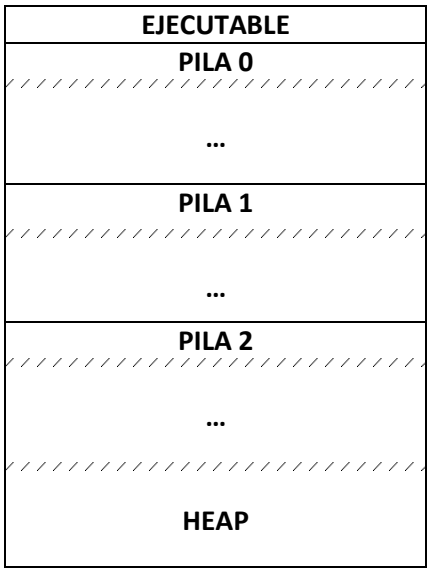
Si yo disparo 3 hebras de ejecución al mismo momento, lo que pasa es que se pierde la noción de pila porque cuando 2 o 3 hebras corren simultáneamente si yo pusiera todo los registros de activación en una sola pila, por ahí el que termina no es el ultimo, sino uno que está al medio, y se empiezan a armar pilas con agujeros. Entonces no se puede poner todo en una pila porque el orden de comienzo no es el inverso del orden de terminación. Las hebras arrancan en un cierto orden pero terminan en cualquier otro.

Entonces ya no puedo tener una pila de registros de activación y la solución conceptual seria hacer una pila inicial hasta que se arrancan las hebras, y cuando se arrancan las hebras crear, una pila por cada hebra de ejecución. Y cada una de las pilas puede crecer o disminuir a su gusto.



Diferentes pilas cada una con su secuencia de ejecución. Y acá si, quien termina el programa es la última definición que es la que está más arriba.

¿Cómo se hace funcionar esto? Poniendo, el ejecutable, la pila 0 con posibilidades de crecer obviamente, luego otra pila. Se los pone lo más lejos posible la una de la otra para que no se superpongan.



Esto en la práctica no se puede hacer en una computadora con memoria con particiones fijas o con segmentos. Solo se puede hacer en un computador sobre un sistema operativo con memoria virtual. Porque con memoria virtual, realmente esas distancias son grandes.

¿Dónde se ponen las pilas? Se espera que el programador cada vez que arranque una hebra diga dónde va a estar su pila. Y si no lo dice el lenguaje elije un valor por defecto. ¿Y como es el valor por defecto? Típicamente en mitades. Si tengo esto, la primera hebra la voy a poner a la mitad, la otra a la mitad de lo que queda, la otra nuevamente a la mitad de lo que queda y así sucesivamente.

En algunos lenguajes aquí aparece una nueva noción de alcance. Porque puede tener sentido que yo quiera variables locales a las hebras que no quiera compartir con otras hebras o puede querer tener variables globales que se compartan. Surgiendo los problemas clásicos de compartir datos en concurrencia.

Por ejemplo C++ en Linux:

```
Static int Z;
Thread int Z;
```

Hay 2 tipos de variables globales. La primera es global a todas las hebras, o la segunda una variable global privada de la hebra. Entonces en el dibujo de arriba quedaría. Como se ve.

Clase 5. Similitudes y diferencias entre variables y objetos

Si tengo un lenguaje tipo algol, puedo tener objetos en la pila u objetos en el heap.

Si tengo un lenguaje dinámico, los objetos van a estar en el heap referenciados por la tabla de símbolos.

Entonces desde el punto de vista del acceso, los objetos y las variables son indistinguibles. O sea tienen la misma estructura de acceso.

Los objetos se diferencian de las variables en su estructura interna y en el uso de esa estructura interna. Una variable de cualquier naturaleza, tiene en su interior datos. Un objeto tiene datos más punteros al objeto.

Cuando un lenguaje tiene un llamado explicito, tiene la parte del programa que hace el llamado, y la parte del programa a la que se llama y hay un vínculo directo rígido y estática.

Cuando es un acceso a través de un objeto, el llamador, llega al objeto, y del objeto vuelve al código ejecutable, al lugar donde está el método.

¿Cómo se usa la cadena estática en relación con el acceso a métodos globales? La cadena estática, solo se usa para acceder solo a variables no a métodos, las funciones están vinculadas estáticamente si no hay objetos involucrados. E involucran un paso por un puntero que está en un objeto si se trata de mensajes a objetos. Pero la cadena estática no existe para la relación entre métodos.

Compatibilidades y conversiones de tipos de datos

Si tenemos una asignación donde son de tipos diferentes, entonces pueden pasar varias cosas.

a := b;

- Si cambia el tipo de a, estamos en presencia de lenguaje dinámico.
- Si cambia el tipo de b, en realidad está mal dicho, no es cierto que cambia el tipo de b. lo correcto es decir que se convierte del tipo de b al tipo de a. O sea hay una conversión. Es lo que pasa si a es float y b es int.
- Son declarados incompatibles y el compilador se niega a hacer la operación.

Aunque uno piense que la conversión se puede hacer y que es fácil y no trae problema, no todos los lenguajes están dispuestos a hacer las conversiones automáticamente.

Si se hace la conversión b al tipo de a, decimos que tiene **conversiones implícitas**. Si pasa que el compilador no la hace automáticamente, decimos que tiene **conversiones explicitas**.

Por supuesto que no se pueden convertir cosas absurdas. Pero de todas maneras los lenguajes con conversiones explicitas dicen yo no hago ninguna conversión en forma automática. Un ejemplo de conversiones explicitas es ADA. Y lenguajes típicos de conversiones implícitas son C y Algol.

Detrás de las conversiones hay ciertos aspectos filosóficos de construcción del lenguaje, que están relacionados con la:

Legibilidad: es legible cuando entiendo fácilmente lo que está escrito en ese lenguaje.

Y la **escribilidad**: es más fácilmente escribible, cuando no tengo que andar escribiendo mucho.

Estas propiedades notoriamente opuestas. La legibilidad se la considera una buena base para la mantenibilidad.

Las conversiones explicitas facilitan la lectura de los programas. Ejemplo:

```
int a;
long b;
float c;
a := b + c;
```

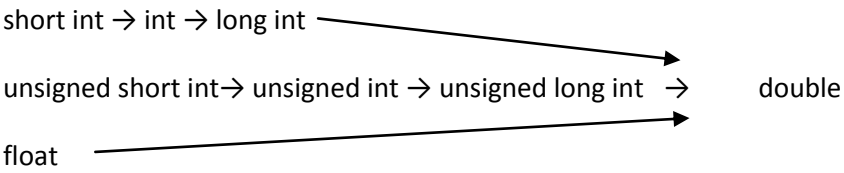
Primero hay que convertir el c que es float a doublé, el b a doublé. Hay que sumarlos en doublé y después antes de hacer la asignación hay que convertirlos a enteros. Hay muchas cosas que pasan y que no se ven en la instrucción.

Esto mismo en ADA seria:

```
a := int( float(b) + float(c) );  // Como ADA no tiene double uso float.
```

Las conversiones implícitas facilitan la escritura de los programas.

Ejemplo en C y todas las conversiones involucradas: (uso → para decir que se convierte implícitamente)



Hay una sola conversión que es algo confusa que es la conversión de long a double y de float a doublé. En C, y en casi todos los lenguajes que tiene esa característica, cualquier operación entre long y float tiene la conversión implícita de cada uno a double. ¿Por qué es eso? Porque un float tiene un exponente que puede no poder representarse en long y un long tiene 32 bits y puede no poder representarse en los 24 de mantisa del float. Entonces no puede convertirse recíprocamente uno en el otro. Así que cualquier relación entre ellos fuerza la conversión a double.

Las conversiones implícitas funcionan más o menos así, del lado derecho de la instrucción siempre se sube a la mayor precisión necesaria. O sea del lado derecho si tengo 2 tipos de diferente precisión, siempre se busca uno más alto que cubra los 2 o uno de los 2. Obviamente int + long se convierte el int a long. Pero luego cuando voy al lado izquierdo, el lado derecho se reduce si es necesario al tipo del lado izquierdo. Entonces se dice que en la asignación manda el tipo del lado izquierdo y del lado derecho manda el tipo de mayor precisión necesaria.

De las conversiones explicitas no hay mucho para decir, hay que escribirlas. Y si están más escritas el compilador te lo dice.

Un aspecto que es primo hermano de las conversiones son las **compatibilidades**. Es decir cuando puedo mezclar 2 tipos en una operación. ¿Puedo meter en una expresión un int mas un carácter sí o no?.

La **compatibilidad por nombre**: se dice que 2 variables son compatibles si tienen el mismo nombre de tipo.

La **compatibilidad por estructura**: se dice que 2 variables son compatibles si se almacenan igual.

Ejemplo en C:

typedef pepe int; //estoy diciendo que pepe es un tipo igual a entero.
pepe a;
int b;

Para C la variable pepe e int son compatibles, porque para C, la compatibilidad es por estructura.

En ADA, la idea de compatibilidad es la siguiente:

type pesos is new float; //estoy diciendo que pesos es un nuevo nombre de tipo.
pesos a;
float b;

Para ADA la variable pesos y float son incompatibles, porque para ADA, la compatibilidad es por nombre. Y si los nombre son distintos entonces no se las puede mezclar, esto es interesante ya que permite construir grupos de variables las cuales no se las puede mezclar.

Por ejemplo:

type dolares is new float;
float c;

Jamás voy a poder sumar dólares con pesos, el compilador me va a decir que está mal. Si las quiero sumar tendría que hacer algo así: a + pesos(c); ó dolares(a) + c;

A ninguna de las dos se las puedo asignar a b porque b es una variable tipo float. Entonces una asignación correcta seria:

b := float(a + pesos(c));
b := float(dolares(a) + c);
b := float(a) + float(c);

Los lenguajes con conversiones explicitas tienen compatibilidad por nombre.

Los lenguajes con conversiones implícitas tienen compatibilidad por estructura.

Tipos de compatibilidades de algol

- DESFERENCING
- DESPROCEDURING
- UNITING
- VOIDING
- ROWING

Desreferencing

Para algol una variable es una referencia a una celda de memoria. En una asignación cualquiera del tipo:

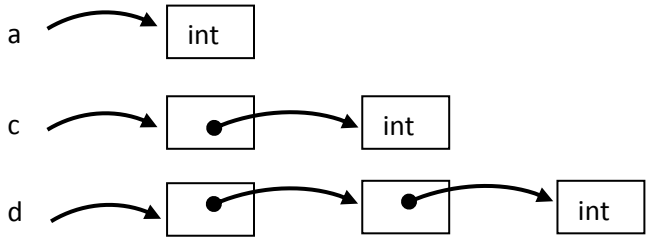
X := Y;

Lo que dice Algol, es el lado izquierdo siempre debe actuar como una **referencia** a una celda. Y el lado derecho debe ser un **valor** almacenable en la celda referenciada por X.

En C, es común hablar del lvalue y el rvalue, cuando se hacen cosas raras en C o C++, por ahí el compilador dice no está usando un valor correcto del lado izquierdo.

Vamos a declarar algunas variables, la palabra ref en Algo es exactamente lo mismo que "*" en C:

En Algol	En C	
int a, b;	int a, b;	Enteros
ref int c, d;	int *c, *d;	Punteros a enteros
ref ref int e, f;	int **e, **f;	Punteros a punteros a enteros



Podemos decir que:

- a y b, son referencias a una celda donde cabe un entero.
- c y d, son referencias a una celda donde cabe la dirección de un entero.
- e y f, son referencias a una celda donde cabe la referencia de otra celda donde cabe la dirección a un entero.

Vamos a poner todas las posibles asignaciones entre una variable de un tipo y otra de otro tipo:

	En Algol	En C	Desreferencing
Copian enteros	a := b;	a := b;	1 extracción de valor
	a := c;	a := *c;	2 extracción de valor
	a := e;	a := **e;	3 extracción de valor
Copian punteros	c := a;	c := &a;	0 extracción de valor
	c := d;	c := d;	1 extracción de valor
	c := e;	c := *e;	2 extracción de valor
Copian punteros a punteros	e := b;	MAL	MAL NO SE PUEDE HACER
	e := c;	e := &c;	0 extracción de valor
	e := f;	e := f;	1 extracción de valor

Cuando hago:
a := b; **b** es la dirección de un entero pero **a** quiere el entero, no la dirección entonces se dice que se hace un **desreferencing**. Que lo vamos a leer como 1 extracción de valor. Entonces la conversión implícita desreferencing implica extracción de valor. Seguir puntero y extraer el valor.

En C hay un desreferencing automático. Esto pasa en casi todos los lenguajes de programación. Cuando tengo una variable del lado derecho y una variable del lado izquierdo, del lado izquierdo tengo una referencia a una celda, del lado derecho tengo que tener un valor que sea compatible con esa celda.

En Algol la idea que se tenía era que las conversiones fueran implícitas, si había que seguir un puntero, que se siguiera. Algol fue un lenguaje extremo en cuanto a la facilidad de escritura y extremo en cuanto a la dificultad de lectura. Porque si yo tengo la instrucción: a := e; yo no sé qué es lo que pasa, tengo que mirar los tipos para darme cuenta. Entonces esta

situación de desreferencing fue completamente abandonada. Ningún otro lenguaje la tuvo nunca más en cuenta y en todos los lenguajes posteriores el residuo que quedó es un desreferencing implícito.

Es decir del lado derecho tiene que haber lo que en C se llama un rvalue, que es extraer el valor compatible con el lado izquierdo.

Uniting

Algol soporta al igual que muchos otros lenguajes un tipo de variable **unión**, estas variables de tipo unión tienen la propiedad de tener una suerte de tipo dinámico ilimitado, ejemplo

union (int, real) x; //la variable x puede adoptar el tipo x y real.

Cuando se declara una variable de tipo unión en Algol, se crea una zona de datos tan grande como la más grande de los dos tipos. En el ejemplo usaría 4 bytes representando el real. Y se le agrega un pedacito que se llama discriminante.

4 bytes	discriminante

Si tengo las siguientes instrucciones:

union (int, real) x;
int a;
real b;
x := a;//se puede convertir de un entero en una unión

int	No usado	int

Lo que pasa es lo siguiente se guarda un entero, después va a haber un pedazo no usado, y se indica al discriminante que lo que hay guardado es un entero.

x := b;//se puede convertir de un real en una unión

real	r

Lo que pasa es lo siguiente se guarda un real, y se indica al discriminante que lo que hay guardado es un real.

A estas operaciones en las que el lado derecho es un tipo componente de la unión, y el lado izquierdo es de tipo unión en algol se los llama operación de **uniting** y es una de las conversiones implícitas previstas en Algol.

De todas maneras en Algol se dice que es una unión segura, el peligro de las uniones es, entender mal lo que tengo almacenado. Si tengo una instrucción del estilo:

b := x;//error

El compilador de Algol se niega a poner una variable tipo unión del lado derecho.

Cuando tengo del lado derecho de una instrucción una variable del tipo unión, debo usar una clausula tipo case:

```
case x in:
    when x int: a:= x + 2;
    when x real: b:= x / 3,5;
endc
```

Esta clausula que se la conoce con el nombre de clausula de conformidad, permite usar las uniones indicando que hacer cuando lo que hay almacenado es entero, y qué hacer cuando lo que hay almacenado es real. Es decir esta clausula de conformidad es un if sobre el discriminante.

No tiene por qué estar las dos clausulas, puede haber una sola.

Rowing

La palabra row es la palabra que usa Algol por arreglo entonces si tengo:

```
int [4..20] z;
Z := 0;//hace que cada valor del componente adquiera el valor 0
```

Es una conversión implícita muy fuerte, porque convierte de un entero a un vector.

Voiding

Algol tenía un tipo llamado void, es un tipo que prospero en C, pero no como tipo sino como puntero a void. El tipo void en algol era el tipo nulo, el tipo que no es capaz de almacenar ningún dato. Si yo digo:

```
void z;
```

Es una variable que no es capaz de almacenar nada, y no va a tener ningún valor. Lo cual es en apariencia ridícula pero esto sirve para completar algunas cosas, en Algol existían varias clausulas, varias situaciones en las cuales la función estaba obligada a devolver un valor y yo no lo podía ignorar. En muchos lenguajes las funciones que devuelven un valor se lo puede ignorar. Ejemplo quien usa lo que devuelve un “printf” en C que es un dato entero que nadie usa.

Entonces si yo tengo una función que devuelve un valor, yo estoy obligado a usarlo para algo, por ejemplo mandarlo a una variable para que lo destruya.

Desproceduring

En algol los procedimientos se los trata, fue el único lenguaje que introdujo eso, después esto cambió a C de otra manera, pero ya se tenía idea en esa época de tener programas genéricos, y para lograr programas genéricos la solución de algol fue la de tener que las funciones tuvieran el mismo tratamiento que las variables. Es decir las funciones en algol se copian. Yo puedo escribir un procedimiento con su definición:

```
proc XX = begin
    …
    …
end
```

Pero también puedo escribir procedimientos sin que tenga ninguna instrucción.

```
proc tmp;

tmp := XX;
XX := YY;
YY := tmp;
```

Donde todos son procedimientos. Si hago esto, estoy intercambiando las instrucciones del procedimiento XX con las del YY. La idea que tenia esto, era hacer programas genéricos donde llamo a una función la llamo **fun** y después copio sobre ese fun la función que me interesa y después la especializo en alguna situación que me interesa.

La idea de especialización de funciones se implementó en algol a través de copia de funciones. ¿Cómo se hace para que esto ande? Todo el acceso a función es a través de punteros. Tengo un conjunto de punteros y cuando a una función la copio, copio el puntero entonces esa función que iba para un lado ahora va para otro.

Esta idea de Algol que las funciones se copien se transformo en la idea de punteros a funciones de C o punteros a Objetos en java.

Desproceduring es la conversión implícita cuando tengo una asignación, donde la izquierda es un tipo numérico y del lado derecho tengo una función, lo que dice algol es, el lado derecho se debe convertir en un valor que le sirva a la celda apuntada por el lado izquierdo y la única forma que tiene una función de convertirse en un numero es ejecutándose.

Conversiones y compatibilidades

Vamos a ver un programa en C:

```
float X,Y;
X = 0.1;
Y = 10.0;
if(X * Y == 1.0) printf(“Que bien”);
else printf(“Que mal”);
```

¿Qué escribe ese programa? Esta pregunta no está completa, requiere saber con qué opción de compilación y con qué bibliotecas se compila el programa, en otras palabras se requiere saber que float y que estructura de almacenamiento tienen las variables X e Y.

Las variables reales se pueden almacenar en los distintos lenguajes de programación en formato IEEE o en BCD. La representación de un número real en un formato IEEE es una representación muy compacta pero tiene un pequeño inconveniente.

La representación IEEE se representa básicamente con $2^3 2^2 2^1 2^0, 2^{-1} 2^{-2} 2^{-3}$… O sea que la representación de un número en IEEE se hace en potencias de 2. Por supuesto tiene la mantisa, se corre, el exponente guarda cuando corrí para la izquierda o la derecha, pero esa es la base.

Si recordamos para convertir a binario un número decimal multiplicábamos por 2, y nos quedábamos con la parte entera por ejemplo 0,375 en binario:

0,375 x 2 = 0,750
0,750 x 2 = 1,50
0,5 x 2 = 1,750
Y el numero terminaba siendo ,011.

Pero ahora veamos el número 0,1 que es un numero que aparece mucho.

0,1 x 2 = 0,2
0,2 x 2 = 0,4
0,4 x 2 = 0,8
0,8 x 2 = 1,6
0,6 x 2 = 1,2
0,2 x 2 = 0,4
0,4 x 2 = 0,8
Y el numero terminaba siendo ,00011001100… periódico

Entonces el 0,1 en decimal no se puede representar de forma exacta en binario. La consecuencia que trae esto es que 10 * 0,1 es 0,99999… pero no 1. Por lo tanto el programa imprime “Que mal”. Consecuencia que tiene esto, es que si hacemos una un formulario de factura y tenemos previsto un casillero de 2 cm, donde vamos a poner 17,30 el lenguaje nos escribe 17,2999… O si se hace un sistema de contabilidad, empiezan a faltar centavos que no se saben donde se perdió.

Nosotros estamos acostumbrados a que 1/3 es periódico = 0,33333… pero no estamos acostumbrados a que 1/10 o 1/100 sea periódico.

La aritmética BCD es una aritmética donde todos los números se representan en binario codificado decimal. Básicamente lo que usa es 4 bits para cada digito. Y los dígitos que se ponen son números decimales. Esta aritmética es más cercana a la costumbre que tenemos al hacer cuenta. Y por lo tanto es una aritmética que nos resulta más natural, pero es más ineficiente porque 4 bit son 16 valores y yo voy a usar 10. Es más complejo sumar en BCD. Pero 0,1 no es periódico.

¿Qué han hecho los lenguajes con esto? ¿Los lenguajes votaron por IEEE o por BCD?

Fortran utiliza una representación pre-IEEE, porque fue una representación que dio lugar posteriormente con algunos cambios a la norma de IEEE.

Cobol optó por BCD.

Pascal por IEEE.

Pero algunos lenguajes como C, es opcional, entonces uno puede tener un programa en BCD o en IEEE, de acuerdo a sus necesidades. Pero todas las variables reales son BCD o IEEE.

Hay un lenguaje en particular que le da la opción al programador variable a variable. Y ese lenguaje es ADA.

type reales_base_10 is new row float delta 0,01// estoy creando un tipo de datos donde hay solo 2 decimales

Para las variables que involucran pesos o moneda en general, este tipo de datos es ideal. Entonces si yo creo:

a : reales_base_10;
b : float;

Estas dos variables son incompatibles. Entonces si yo quiero sumar a y b, ¿En que las queres sumar? ¿En base 10 o 2?

Si las quiero sumar en base 2:

float(a) + b;//suma IEEE

Si las quiero sumar en base 10:

a + reales_base_10(b);//BCD

¿Quién es el culpable de la periodicidad de 0,1?

Voy a pensar en $\frac{1}{n}$ y anotamos:

2	3
4	5
6	7
8	9

- $\frac{1}{2}$ es exacto en base 10 es 0,5 y es exacto en base 2 es 0,1.
- $\frac{1}{3}$ es periódico en los 2 entonces no me engaña, en cualquiera de las dos aritméticas va a dar periódico entonces ya estoy prevenido.
- $\frac{1}{4}$ es exacto en las 2 representaciones.
- $\frac{1}{8}$ es exacto en las 2 representaciones.
- $\frac{1}{9}, \frac{1}{6}, \frac{1}{7}$ son periódicos en las 2 representaciones.
- $\frac{1}{5}$ es el desgraciado, porque da una representación exacta en base 10 y periódica en base 2.

Entonces cuando el divisor es 5 o cualquiera de sus múltiplos dan periódicos en aritmética IEEE.

Copiado hasta el final de la clase 5. Faltan clases teóricas de la cursada.