RESUMEN DE MAK – LENGUAJES Y COMPILADORES (Primer Cuatrimestre)

Autor: Ariel Nader – arielnader@gmail.com

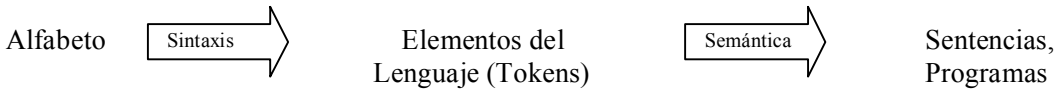
Definiciones Iniciales

Los lenguajes de programación tienen tres aspectos fundamentales:

- Sintaxis: Es el conjunto de reglas que especifican la composición del programa a partir de letras, dígitos y otros caracteres. Esto no nos dice nada acerca del significado de cada sentencia, solo indica si una sentencia determinada es válida dentro del lenguaje o no. Esta compuesta por dos elementos:
  - Léxica.
  - Gramática.

- Semántica: especifica el significado de un programa escrito de forma válida bajo las reglas de sintaxis. Definen cual es el efecto de cada una de las sentencias y el programa completo.

- Pragmática: Parte del estudio de los lenguajes que se dedica a todo lo que tiene que ver con las cuestiones de construcción. Implementación, velocidad, versiones, etc.



Sobre el conjunto de caracteres del alfabeto, podemos aplicarle reglas de sintaxis para obtener los elementos del lenguaje válidos. Ejemplos de elementos son la palabra "while", las variables, las constantes, la palabra "int", etc. A este conjunto, se le aplican las reglas de semántica para definir las sentencias y los programas válidos para el lenguaje.

Binding y Variables

El binding o ligadura es el momento exacto en el que conoce un atributo o propiedad de un elemento de cierto lenguaje. Se entiende por elemento de lenguaje a variables, identificadores, constantes, funciones, procedimientos, etc. De esta forma, el binding de una variable es el concepto que define el momento en el que se conoce una propiedad determinada de una variable (es decir, el momento preciso en el que una propiedad de una variable está definida). El binding es un concepto central en la definición de la semántica de los lenguajes de programación.

Cada variable tiene un nombre y este es utilizado para que pueda ser referenciada. Las variables tienen cuatro atributos que podemos estudiar en términos de binding:

Tipo: Es la especificación del conjunto de valores que puede tener una variable, junto con las operaciones en las que puede intervenir. Cuando se crea un lenguaje, se definen un grupo de tipos de datos de los que puede ser una variable (ej: int, char, bool, etc). En algunos lenguajes el programador puede definir nuevos tipos.

Los tipos pueden enlazarse:

- Estáticamente: por ejemplo, en C una variable definida como "int var" siempre será int. Esto se define en tiempo de compilación.
- Dinámicamente: por ejemplo, en un compilador de BASIC determinado una variable definida como "Dim var" puede en un momento albergar un valor numérico y mas tarde un string, esto hace que el tipo de la variable cambie. Otro ejemplo ocurre en el lenguaje APL. Así, el tipo va a depender del flujo de ejecución del programa.

Valor: Es el contenido de la variable en un determinado momento. Se representa codificado por medio de bits y, según el tipo de la variable, esa representación tiene un significado distinto. Este valor puede ser modificado por una operación de asignación.

Los valores pueden enlazarse:

- Estáticamente: para el caso de las constantes simbólicas, el valor nunca cambia a lo largo de la ejecución. Por ejemplo, "const int var = 3". Esto se establece en tiempo de compilación.
- Dinámicamente: Es el caso de cualquier variable común que por medio de una asignación su valor cambia. Todo esto depende del flujo de ejecución.

Alcance: Es el rango de instrucciones del programa en el cual es conocida la variable.

El alcance puede enlazarse:

- Estáticamente: En este caso, esta perfectamente definido cuales instrucciones pueden acceder a la variable al momento de la compilación. Ejemplo en C:

```
{
        int x = 0;
        x = x + 1;
}
y = x;
```

  Este código da error porque la variable x no existe en el momento de asignarla a la variable y. Podríamos decir, que el alcance de la variable puede delimitarse con terminales de la estructura léxica del programa (es decir, las llaves {}).

- Dinámicamente: Para este enlace, el alcance se define en el momento de la ejecución del programa. Ejemplo en GW BASIC:

```
10:     A = 10
20:     INPUT X
30:     IF X > 0 THEN GOTO 50
40:     B = 7
50:     C = A + B
```

  Lo que ocurre aca es que la variable B puede ser conocida o no en la línea 50 dependiendo del valor ingresado por teclado en la línea 20. Si el usuario ingresa –1 el flujo de programa pasa por 40 y la variable B es dimensionada en memoria. Pero si el usuario ingresa 1, entra en el IF y este hace un salto a la línea 50 de tal forma que nunca se dimensiona en memoria la variable B y en la línea 50 intenta utilizarla sin antes crearla lo que resulta en un error de ejecución.

Almacenamiento / Tiempo de Vida: Es el momento en el cual un área de memoria es asignada a la variable para que pueda contener un valor.

El almacenamiento puede enlazarse:

- Estáticamente: en este enlace, se conoce la posición de memoria que ocupara cada variable al momento de la compilación ya que al inicio de ejecución se reserva toda la memoria necesaria total. Lenguajes como FORTRAN y COBOL trabajan de esta forma. Por ejemplo, la variable "precio" al momento de compilar siempre ocupara la dirección 03AF6x0 en la memoria. Esto no permite la recursividad ya que no puedo tener dos variables "precio" al mismo tiempo ya que hacen referencia a la misma celda de memoria.
- Dinámicamente: La mayoría de los lenguajes no definen el lugar físico de sus variables hasta tanto no esté el programa en ejecución y el bloque que contenga la variable no sea activado. De esta forma se aprovecha mas la memoria y se permite la recursividad.

Entonces, **podemos decir que un binding es estático si está establecido antes del momento de la ejecución del programa y no puede ser cambiado mas tarde, y que es dinámico si se establece en tiempo de ejecución y puede ser cambiado de acuerdo a algunas reglas especificadas por el lenguaje.**

Atributo del Binding de Variable	Estático	Dinámico
Valor	Constantes	Mayoría
Tipo	Mayoría	APL, LISP, J, SMALLTALK
Alcance	Mayoría	BASIC, APL, J
Almacenamiento	FORTRAN, COBOL	Mayoría

Clasificación de los Lenguajes de Programación

Los lenguajes se clasifican en tres grandes grupos según los siguientes criterios:

- Si tiene almacenamiento estático: → LENGUAJE ESTATICO
- Si tiene almacenamiento dinámico:
    - Si tiene tipo y alcance estáticos: → LENGUAJE TIPO ALGOL / ORIENTADO A LA PILA
    - Si tiene tipo o alcance dinámico: → LENGUAJE DINAMICO

Modelos de Ejecución

Existen dos modelos:

- Compilada: toma un programa escrito en un determinado lenguaje y lo traduce a otro lenguaje, sea este directamente ejecutable o no. Es decir, no necesariamente el lenguaje resultante se puede ejecutar directamente por una CPU. Ej: el EXE que genera un programa en C++ o el código IL de .Net.
- Interpretada: el interprete toma un programa escrito en determinado lenguaje y ejecuta cada instrucción directamente.

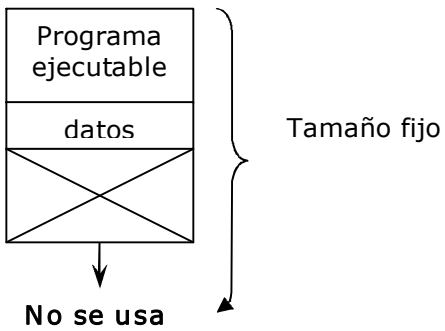
No existe un lenguaje que utilice 100% uno de los modelos pero la mayoría de los lenguajes tipo Algol se acercan al modelo compilado y la mayoría de los lenguajes dinámicos se acercan al modelo interpretado.

Uso de Memoria

- Lenguajes Estáticos: La memoria que el programa necesita es reservada antes del inicio de la ejecución. Cada variable tiene una posición prefijada en la memoria y mantiene su espacio durante todo el tiempo que se esté ejecutando el programa. Estos lenguajes no permiten recursión.
    - Tipo: estático;
    - Alcance: estático;
    - Almacenamiento: estático;

  Ejemplos: COBOL, FOLTRAN.

  Estos lenguajes hacen una asociación rígida entre el nombre de la variable y la dirección de memoria que ocuparán. Por ejemplo, la variable "precio" es reemplazada por la dirección 59A0F al compilarse. Estos programas deben ejecutarse siempre en el mismo lugar de memoria y es por esto que siempre se sabe cuanta memoria ocupan.



  Si tenemos la siguiente asignación donde se utilizan 3 variables enteras:
  A = B + C

El compilador traduce las variables a una dirección fija en memoria:

A → F123
B → AB11
C → 1234

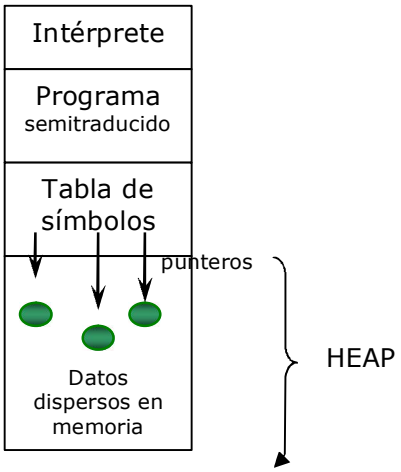
Entonces compila la asignación al siguiente código asembler:

MOV AX, AB11
ADD 1234, AX
MOV F123, AX

- Lenguajes Dinámicos: Tienen un uso de memoria impredecibles. Permiten la recursividad
  - Tipo: dinámico;
  - Alcance: dinámico;
  - Almacenamiento: dinámico;

Ejemplos: LISP, PROLOG, APL, SNOBOL4.

En estos lenguajes una variable puede cambiar de tipo tan solo con recibir una asignación de un valor distinto al tipo original que contenía. Por ejemplo, se puede comenzar asignando a la variable "precio" con un valor de 10 y mas adelante a la misma variable asignarle el string "diez". Esto provoca un cambio de tipo de dato de la variable. De esto podemos deducir que en tiempo de compilación, no se puede determinar que operaciones están permitidas para cierta variable, ya que al compilar la operación "precio / 5" no sabemos si esta variable va a ser INTEGER o STRING (en el primer caso se permite la división, en el segundo no).
Para implementar algo asi, necesitamos tener punteros al descriptor de la variable "precio" dentro de la TABLA DE SIMBOLOS. Esta tabla es un listado de descriptores que contienen entre otras cosas, el nombre de la variable (si, el nombre que se le da en el código, en este caso "precio"), el tipo(si es INTEGER, STRING, etc en este momento) y un puntero al sector del HEAP donde se encuentran los datos de la variable.
El HEAP es un sector de memoria dedicado a contener los valores, clases y estructuras de las variables dinámicas (profundizaremos mas adelante en estos conceptos).
Algo para destacar es que por lo general, los lenguajes estáticos y de pila se compilan mientras que los dinámicos se interpretan. Esto quiere decir que necesitan de un programa interprete que esté ejecutando en memoria para ser ejecutados.



Si tenemos la siguiente asignación donde se utilizan 3 variables enteras:
A = B + C

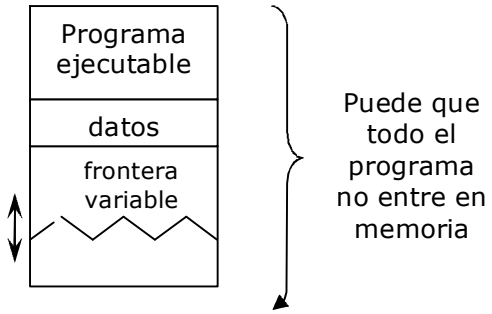
El interprete traducirá la asignación a una serie de operaciones como las siguientes:

- Buscar en Tabla de Símbolos a B
- Buscar en Tabla de Símbolos a C
- Llamar rutina de suma enteros
- Buscar en Tabla de Símbolos a A
- Almacenar el valor resultante en A

- Lenguajes basados en Pila / Tipo ALGOL: No se puede asegurar el uso de memoria de los programas de este tipo, pero se puede predecir. El uso de memoria sigue una disciplina LIFO (tipo como una pila).
  - Tipo: estático;
  - Alcance: estático;
  - Almacenamiento: dinámico;

  Ejemplos: ALGOL, C, PASCAL, ADA, MODULA.

  Las variables no tienen predefinido el lugar exacto donde serán contenidas en memoria, en vez de eso, cada variable tiene fijo un offset desde el bloque que le corresponda. Este bloque es como un segmento de datos asociado a una unidad de ejecución (función, procedimiento o bloque de código) y no tiene un lugar fijo en memoria. Gracias a esto, permiten la recursividad. Por ejemplo, la función sumar() se ejecuta en un lugar X en memoria. Para esto, se genera un bloque asociado a dicha función donde estarán contenidas cada variable declarada dentro de la misma. Entonces, la variable "precio", que está declarada en la función, es reemplazada al compilar por un OFFSET fijo desde el comienzo de dicho bloque. De esta forma, si la misma función sumar() se llama a si misma, un nuevo bloque se reserva en otro sector de memoria y la nueva variable "precio" tendrá el MISMO OFFSET en cada ejecución pero la BASE del BLOQUE será distinta permitiendo poder ser diferenciadas. A este bloque se lo llama **REGISTRO DE ACTIVACION**. Hablaremos de este concepto mas adelante.



Unidad o Bloque

Se define como un conjunto de instrucciones delimitadas de forma explicita donde se permite declarar variables locales. Las instrucciones dentro del bloque conocen y pueden referenciar a las variables que allí dentro se declaran, pero una vez que el bloque finaliza, estas variables dejan de existir y no son reconocidas por el resto del programa.

Clasificación:

- Anónimos: No tienen un nombre, simplemente se entra en el debido al flujo de instrucciones, en otras palabras, la ejecución llega al lugar donde se encuentra declarado. Debido a que no puede ser llamado, no es recursivo.
- Con Nombre: Tienen nombre, son funciones, procedimientos, etc. Son invocados por una instrucción y pueden ser recursivos.

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()
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;
```

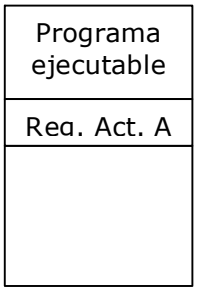
Es importante repasar la definición de Unidad o Bloque para saber cuando estamos hablando de uno y cuando no. En los cuatro casos anteriores, la delimitación del bloque era explicita, pero en el último no se permite declarar variables dentro y esto hace que no sea considerado bloque.

Registro de Activación

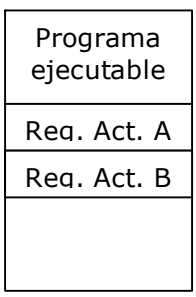
Los registros de activación son una porción de memoria reservada para los datos que manipula una unidad o bloque en ejecución. Cuando en el programa se llama a una función, inmediatamente se reserva en memora el espacio donde contendrá las variables locales de la función, los parámetros de entrada y 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.

Supongamos la secuencia de llamados de los siguientes bloques A → B → C. Entonces diagramemos lo que va ocurriendo a cada momento:

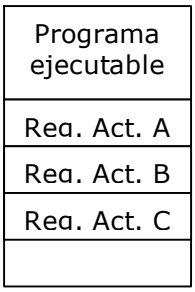
1) En memoria está alojado el código del programa. Como A está ejecutando también se tiene en memoria el registro de activación de A con todas sus variables locales.



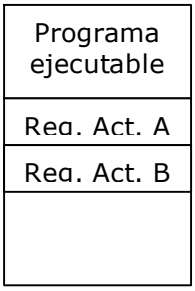
2) Luego A llama a B, con lo cual el registro de activación de B es cargado.



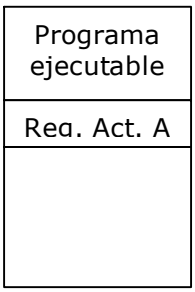
3) Y luego B llama a C.



4) Finalmente C retorna y se elimina en memoria su registro de activación.



5) Lo mismo para B cuando retorna.



Esto demuestra claramente que los registros de activación tienen un comportamiento de tipo LIFO (Last Input First Output) y de ahí del nombre de "lenguajes orientados a la pila". Se dice que cuando el registro de activación de una unidad se carga, adquiere direccionamiento.

Como habíamos dicho, cada variable se remplaza por un offset fijo en el momento de la compilación, cuya base viene dada por la dirección de comienzo del registro de activación que la contiene. Esta base es guardada generalmente en el registro BP (o EBP). Entonces supongamos que tenemos las siguientes variables y que el compilador le dio esas distancias de offset que se aclaran en el cuadro:

Variable	Offset
Z	10
X	20
Y	30

La instrucción Z = X + Y se compila de la siguiente manera:

```
MOV     R1, [BP + 20]          # Se carga en el registro R1 el valor de X
ADD     R1, [BP + 30]          # Se suma al registro R1 el valor de Y
MOV     [BP + 10],  R1         # Se almacena en Z el valor resultante
```

Esos offset son asignados al momento de la compilación y quedan fijos en el código del programa. De esta forma podemos ver claramente que el nombre de la variable no queda disponible en el momento de ejecución en los lenguajes de tipo Pila. Esto solo ocurre en los lenguajes Dinámicos.

Cadena Dinámica

Antes de explicar este concepto, vamos a explicar la problemática. En cada momento, el programa mantiene al registro BP apuntando a la base del registro de activación activo. Cuando se produce un llamado a otro bloque, antes de realizar el salto al código correspondiente, se debe apuntar al nuevo registro de activación. Este nuevo registro estará contiguo al registro anterior con lo cual, para apuntar al nuevo se le debe sumar al BP el tamaño del registro anterior. Las instrucciones necesarias para hacer esto las construye el compilador ya que conoce el tamaño del registro de activación actual.

```
ADD     BP, (TAMAÑO R. A. ACTUAL)            # Se suma el tamaño del R. A. Actual.
```

Mas adelante, con la ejecución del programa llega el momento de retornar de una función. Lo que se debe realizar en ese momento es volver atrás el BP, es decir, apuntar al registro de activación anterior. Pero el problema es que en este caso, una función no puede saber por quien ha sido llamada, con lo cual, no sabría a cuanto restarle al BP. Para solucionar esto, el compilador reserva en un lugar fijo de cada registro de activación la dirección del registro de la unidad que lo llamo.

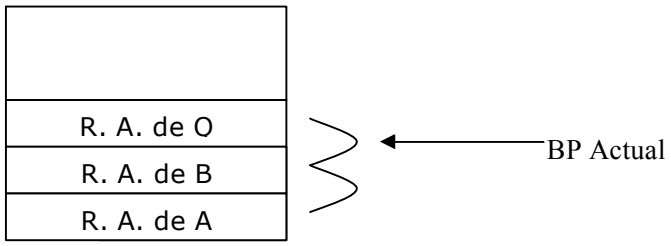
Este comportamiento genera una sucesión de punteros que va desde el registro de activación activo hasta el primero, pasando por todos dentro de la pila de llamadas. A esta cadena de punteros se le llama **CADENA DINAMICA**.

Entonces, ampliamos el código del llamado con lo explicado anteriormente:

```
MOV     R7, BP                          # Guarda en un registro cualquiera, el valor actual de BP.
ADD     BP, (TAMAÑO R. A. ACTUAL)       # Se suma el tamaño. Esto genera cambio de contexto
MOV     [BP + 4], R7                    # Guarda el valor del BP anterior en el R.A. actual, posición 4
....
....                                    # Se ejecuta el código de la función llamada
....
MOV     BP, [BP + 4]                    # Vuelvo a apuntar al BP anterior
```

El compilador decide a que offset fijo va a guardar el puntero BP anterior, y lo respeta para todos los registros de activación (en este ejemplo de código, el offset es de 4).

A continuación realizamos una representación de la cadena dinámica para la secuencia de llamados A → B → Q al momento de ejecutar Q. Omitiremos el código del programa para simplificar el dibujo, e invertiremos el sentido de la memoria.

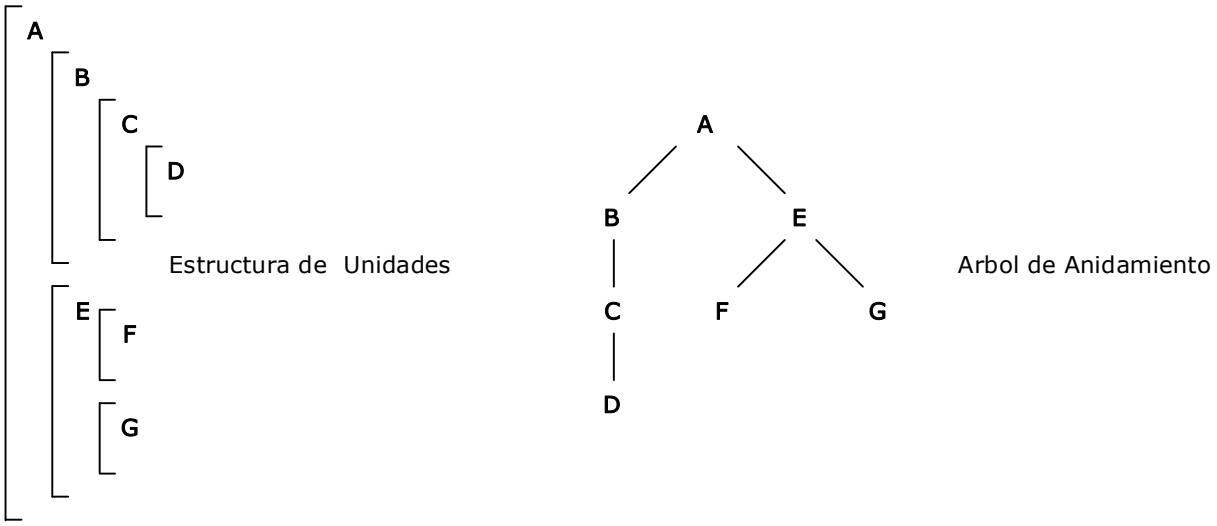


En el dibujo se ve claramente como cada registro de activación tiene en un offset fijo un puntero al registro anterior formando una cadena de punteros, y el BP actual apuntando al registro activo.

Estructura de Unidades

Los lenguajes de tipo Algol tienen una estructura que tiene por objetivo poder dividir el código en unidades y controlar el ámbito de las variables. Esto es un anidamiento estático de unidades, las define el programador y no varia

luego de la compilación. Las unidades pueden estar anidadas hacia adentro o independientes. Esto puede ser representado con las siguientes imágenes:

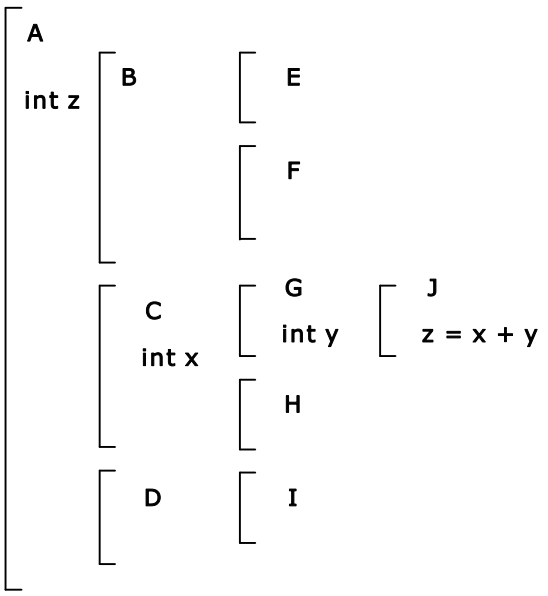


En la primer figura se muestra un programa que tiene una unidad principal A. Este contiene dos unidades B y E. A su vez, B contiene la unidad C y este contiene a D. Por otro lado, la unidad E contiene las unidades F y G. Esta misma estructura puede representarse con el árbol de la segunda figura.

En cada unidad se pueden declarar variables. Una variable declarada en la unidad A tiene nivel 0, una declarada en el B o E tiene nivel 1 y así con las demás unidades, la notación para describir el nivel de anidación de una unidad es "nivel(U)" para la unidad U. Cada unidad puede manipular las variables que se declaran en ella de manera local y las variables que se declaran en unidades donde están contenidas de manera global. Por ejemplo, una variable declarada en B es local a B y global a C y D. En el caso que una variable tenga el mismo nombre que otra global a la unidad, se refiere a la variable local sin tener acceso a la global. Esto quiere decir que la variable local enmascara a la variable global.

El árbol de anidamiento es una estructura que construye el compilador al momento de compilar, la utiliza para tal fin y una vez que termino el proceso de compilación la destruye.

Ahora bien, supongamos un programa con la siguiente estructura:



Este no es lenguaje C. Vemos la estructura y destacamos que en la unidad A se declara la variable "z", en la C la variable "x", en la G la variable "y", mientras que en la unidad J se opera con dichas variables.

Si las variables fueran declaradas en la unidad J, la operación de suma seria sencilla de compilar ya que se debería reemplazar cada una con el [BP + offset] como se mostró anteriormente. Pero la cuestión es que estas variables no se encuentran ubicadas en el registro de activación actual, con lo cual obliga al programa a buscarlas en los anteriores.

Para tal fin, el diseñador del compilador debe establecer una regla de alcance y seguir dicha regla al compilar. Un ejemplo seria: "Busque las variables en el entorno local, sino están ahí busque en la unidad padre y así sucesivamente. Para ello, debemos saber a que distancia esta cada variable(dentro del árbol de anidamiento) según la unidad donde se encuentra la operación. Para este caso tenemos:

Variable	Arcos
Z	3
X	2
Y	1

El concepto de arco es la cantidad de saltos atrás que debe hacer para encontrar la variable dentro del árbol de anidación. Vemos que para "z" desde la unidad J se debe pasar por G, C y A para encontrarla, para "x" por G y C y para "y" solo por G. Hay que tener en cuenta que cada variable dentro de su registro de activación tiene un offset fijo, con lo cual hay que tener en cuenta las dos cuestiones a la hora de compilar, el offset y los arcos. Vamos a escribir un pseudo código en assembler que de cómo se compilaría la suma y mas adelante lo completaremos:

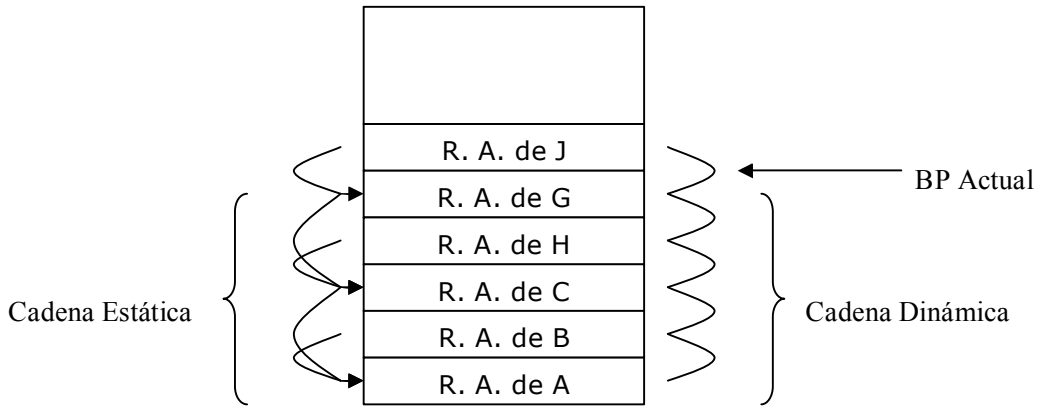
```
MOV     R1, 2 arcos + [BP + 20]                 # Se carga en el registro R1 el valor de X
ADD     R1, 1 arco + [BP + 30]                  # Se suma al registro R1 el valor de Y
MOV     3 arcos + [BP + 10],  R1                # Se almacena en Z el valor resultante
```

Cadena Estática

Como vemos, para obtener el lugar de memoria de una variable se utilizan los arcos. Para esto, el compilador arma una estructura para implementar las búsquedas que realizó en el árbol de anidamiento pero que en ejecución ya no tiene. Esto es una cadena de punteros donde cada registro de activación apunta a la base de su padre. Esta cadena se llama **CADENA ESTATICA**. Entonces cada registro de activación tiene almacenado un puntero (en un offset fijo) que apunta al registro de activación padre dentro de la estructuras de las unidades.

Para la siguiente secuencia de llamados con la estructura anteriormente vista, se muestran los punteros:

A → B → C → H → G → J



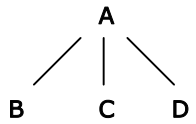
Entonces, para obtener la variable "x" desde J dijimos que necesitaba dos arcos, esto es mover dos a través de la cadena estática y a un offset de 20 es encontrada (dentro del registro de activación de C).

Bueno, ahora si, podemos escribir el código assembler de la suma ejecutada en el módulo J asumiendo que el puntero de la cadena estática se encuentra en un offset fijo de dos dentro del registro de activación.

```
MOV     R4, BP          # Guarda el BP en un registro cualquiera para no perderlo

MOV     BP, [BP + 2]    # Baja dos veces en la cadena estática para encontrar a X
MOV     BP, [BP + 2]
MOV     R1, [BP + 20]   # Guarda en el R1 el valor de X (offset de 20 dentro de su registro de activación)

MOV     BP, R4          # Vuelve al contexto de la unidad J

MOV     BP, [BP + 2]    # Baja una vez en la cadena estática para encontrar Y
ADD     R1, [BP + 30]   # Le suma a R1 el valor de Y (offset de 30 dentro de su registro de activación)

MOV     BP, R4          # Vuelve al contexto de la unidad J

MOV     BP, [BP + 2]    # Baja tres veces en la cadena estática para encontrar a Z
MOV     BP, [BP + 2]
MOV     BP, [BP + 2]
MOV     [BP + 10], R1   # Guarda el resultado en Z (offset de 10 dentro de su registro de activación)
```

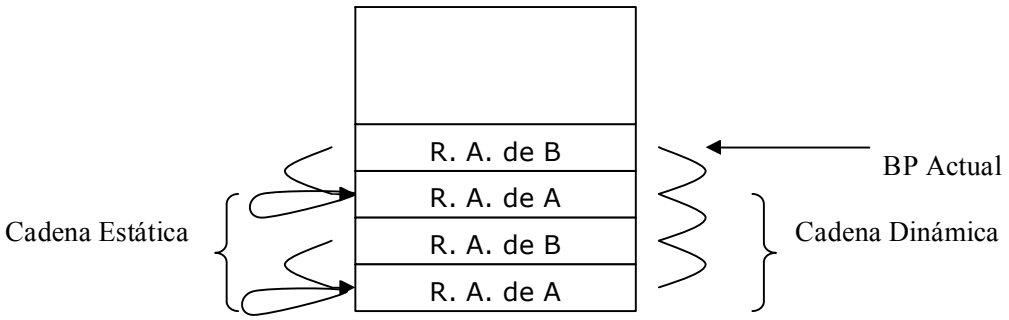
Aclaraciones:

- El lenguaje C no tiene anidación de unidades, solo tiene 2 niveles:

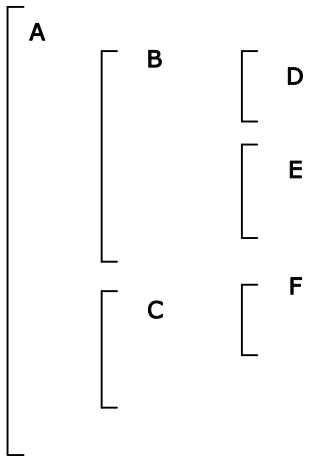


- El puntero de cadena estática de la unidad padre A apunta a si mismo.

- Cada unidad en la cadena estática apunta al padre que se llamó por última vez. Es decir, para la secuencia A→ B → A → B el diagrama es el siguiente:



Para el caso de los llamados entre unidades, una función es global para si misma y para las restantes funciones salvo que sea su padre. Es necesario hablar de la visibilidad entre funciones. Supongamos que tenemos esta estructura:



Las unidades se consideran declaradas en la unidad que es padre. De esta forma, en A se declaran las unidades B y C, en B las D y E y en C la F. De esta forma, deducimos que la unidad A puede llamar a la unidad B porque es local en A. Ahora bien, si estoy ejecutando en la unidad B e intento una llamada a la unidad C, el compilador no va a encontrar en primera instancia a dicha unidad en el entorno local, así que seguirá la cadena estática según indica la regla de alcance hasta encontrarla, con lo cual, baja hasta A y encuentra declarada la unidad C. Observe que solo se hizo una bajada a través de la cadena estática, lo que significa que es Global de Distancia 1. La lógica es la misma para el caso de que D quiera llamar a C, pero la distancia es 2, así que C con respecto a D es global a distancia 2. En el caso de una llamada recursiva, si E quiere llamar a E, al compilar no va a encontrar la declaración de E en su entorno local así que bajará uno por la cadena estática y lo encuentra en B, con lo cual esta llamada es global de distancia 1 (siempre que es recursiva es de distancia 1). Y si B intenta llamar a F? Este caso no es posible y el compilador arrojará un error porque al no encontrar a F en su entorno local baja hasta A y tampoco lo encuentra pero no puede bajar mas ya que estamos en el nodo raíz.

Resumiendo cada caso:

- A → B es local (L);
- B → C es global de distancia uno (G1);
- D → C es global a distancia dos (G2);
- E → E es global de distancia uno (G1 o R de recursiva), todos los llamados recursivos son de esta categoría;
- B → F no se puede realizar;

Para ayudarnos al intentar saber que unidad puede llamar a cual otra, se crea una matriz de llamados:

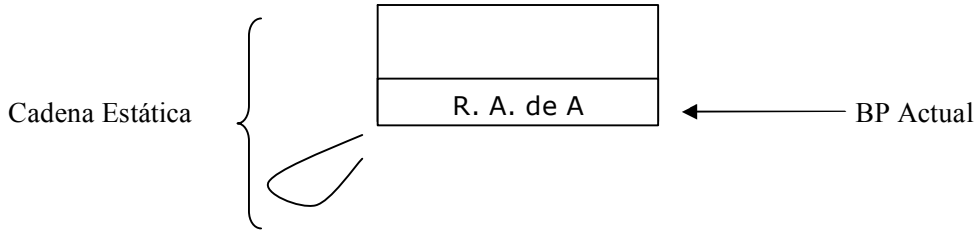
	A	**B**	**C**	**D**	**E**	**F**
A	R	L	L	-	-	-
B	G2	R	G1	L	L	-
C	G2	G1	R	-	-	L
D	G3	G2	G2	R	G1	-
E	G3	G2	G2	G1	R	-
F	G3	G2	G2	-	-	R

Esta matriz no la crea el compilador, solo es una ayuda para nosotros al intentar estudiar el tema.

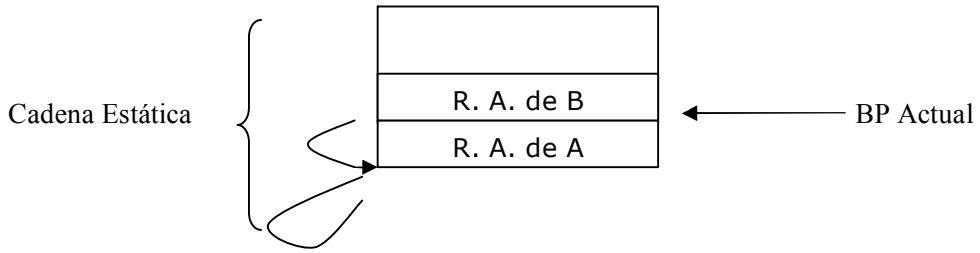
Ahora que sabemos que unidad puede llamar a cual otra, podemos aclarar como es que se construyen los punteros de la cadena estática. Básicamente, el principio que se sigue es que la cadena estática se forma siguiendo la misma cadena estática preconstruida. Veamos un ejemplo, con la siguiente secuencia de llamados:

A → B → C → F → C

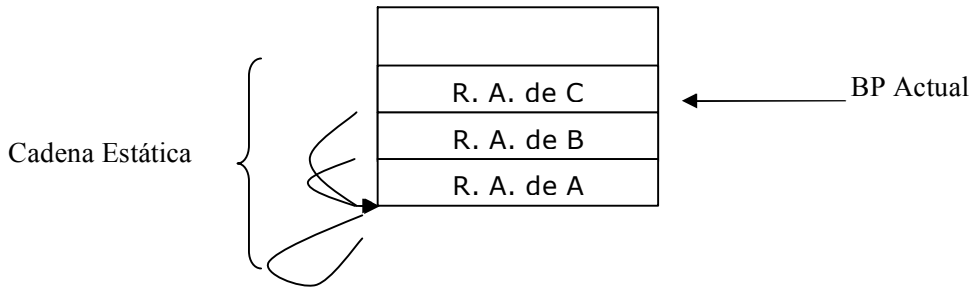
1) Comienza la ejecución de A, en memoria se encuentra un único registro de activación. A continuación el dibujo (por razones prácticas no dibujaremos la cadena dinámica).



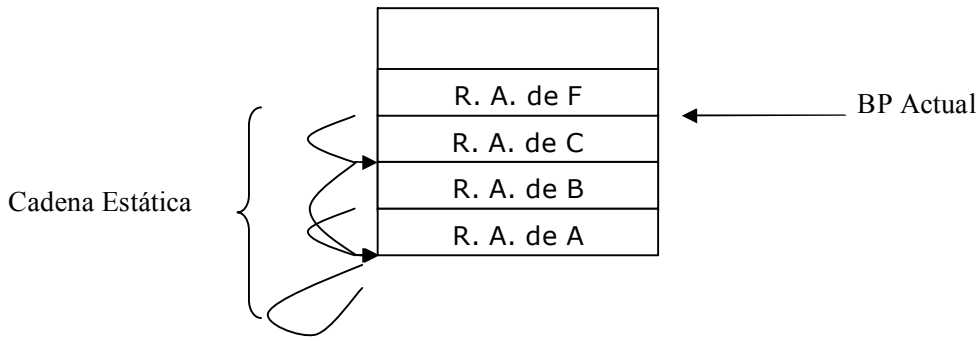
2) A → B: Al encontrar el llamado, el compilador escribe las sentencias que dimensionan el registro de activación de B. Luego de ello, vuelve a apuntar al R. A. de A y consulta el árbol de anidamiento. Como encuentra que B es local a A, toma la dirección de BP de A y la guarda en el puntero de cadena estática de B.



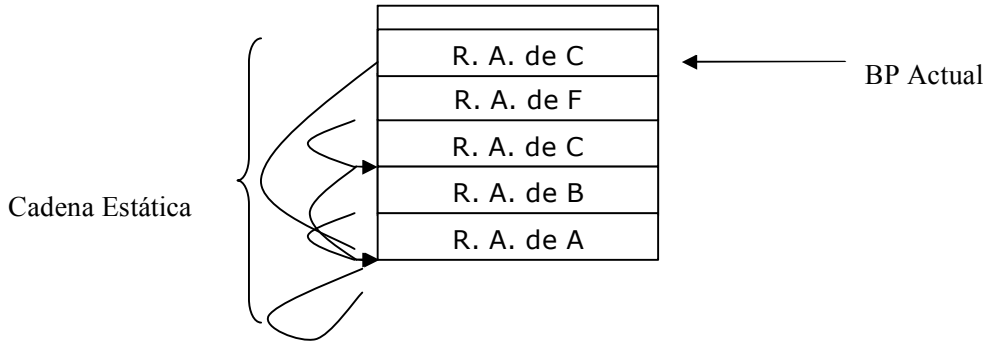
3) B → C: Al encontrar el llamado, el compilador escribe las sentencias que dimensionan el registro de activación de C. Luego de ello, vuelve a apuntar al R. A. de B y consulta el árbol de anidamiento. Como encuentra que C es global a distancia 1 con respecto a B, entonces baja una posición en la cadena estática hasta el R. A. de A y copia esa dirección que apunta a ese R. A. en el puntero de cadena estática de C.



4) C → F: Al encontrar el llamado, el compilador escribe las sentencias que dimensionan el registro de activación de F. Luego de ello, vuelve a apuntar al R. A. de C y consulta el árbol de anidamiento. Como encuentra que F es local a C, toma la dirección de BP de C y la guarda en el puntero de cadena estática de F.



5) F → C: Al encontrar el llamado, el compilador escribe las sentencias que dimensionan el registro de activación de C. Luego de ello, vuelve a apuntar al R. A. de F y consulta el árbol de anidamiento. Como encuentra que C es global a distancia 2 con respecto a F, entonces baja dos posiciones en la cadena estática hasta el R. A. de A y copia esa dirección que apunta a ese R. A. en el puntero de cadena estática de C.



Lo importante a destacar en todo este procedimiento es que según la cantidad de distancia que una unidad sea global a otra es igual a la cantidad de arcos que se debe bajar desde el R. A. de la unidad llamadora a través de la cadena estática preconstruida para obtener que R A. se debe almacenar en el puntero de cadena estática del R. A. de la unidad llamada.

A continuación se presenta la implementación de este procedimiento en código assembler para la llamada F → C y suponiendo que el puntero de cadena estática está a 3 y el de cadena dinámica está a 4.

```
MOV     R6, BP          # Guarda el BP que apunta a C en un registro cualquiera para no perderlo

MOV     BP, [BP + 4]    # Baja al bloque del llamador (es F)

MOV     BP, [BP + 3]    # Baja dos veces a través de la cadena estática. Esto es asi porque C es global
MOV     BP, [BP + 3]    # a dos de F. Si fuera global a 1, el compilador solo pone una sola de estas
                        # sentencias. Si fuera local, no pondría ninguna.

MOV     [R6 + 3], BP    # Como habíamos guardado en R6 el puntero al bloque llamado, lo podemos utilizar de
                        # base para sumarle el offset donde se encuentra el puntero de cadena estática,
                        # y en el guardarle el puntero a la unidad A (que es la actual).

MOV     BP, R6          # Vuelvo a apuntar al bloque llamado para continuar la ejecución
```

Por último vamos a aclarar un detalle. Sabemos que la unidad B puede llamar a la C, pero lo que hay que saber es que no puede acceder a sus variables. Esto es porque para que B acceda debe ir hacia atrás una vez por la cadena estática para encontrar el registro de activación de A que es donde está declarada C. Por eso B accede a llamar a C pero no puede acceder a sus variables, porque nunca llega a pararse sobre su registro de activación (pero si accede al registro de A, por eso puede ver sus variables y la declaración de la unidad C).

Clasificación de las variables:

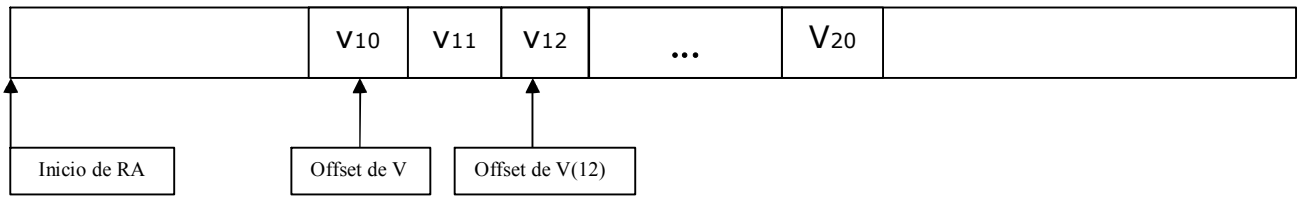
Ya que conocimos la clasificación de los lenguajes de programación, ahora vamos a clasificar las variables con el mismo criterio.

- **Estáticas:** tienen lugar y tamaño fijo en toda la ejecución del programa. Se encuentran en el código en una posición absoluta en memoria. Por lo general no se encuentran en los registros de activación. Son propias de los Lenguajes Estáticos como Cobol y Foltran.

- **Semiestáticas:** tienen lugar variable en distintas ejecuciones y tamaño fijo. Son propias de los Lenguajes basados en la Pila. El registro de activación que las contiene varia de lugar, pero lo que no varia es el offset de la variable dentro del mismo.

  Un arreglo de límites fijos es de este tipo. Por ejemplo: int v(10 to 20)

  El compilador ubica esta variable en el registro de activación, sabe donde empieza y donde termina porque tiene los límites especificados al momento de la definición (almacena todos los elementos del array de forma contigua). Esto se muestra en la siguiente imagen para el array del ejemplo anterior



Al momento de compilar un acceso a un elemento del array, el compilador genera una fórmula para calcular el offset del elemento solicitado. Supongamos que se hace una asignación como la siguiente: v(i) = 2. Primero el compilador averigua el offset con la formula

```
Dv(i) = Dv + (i – 10) * TE
```

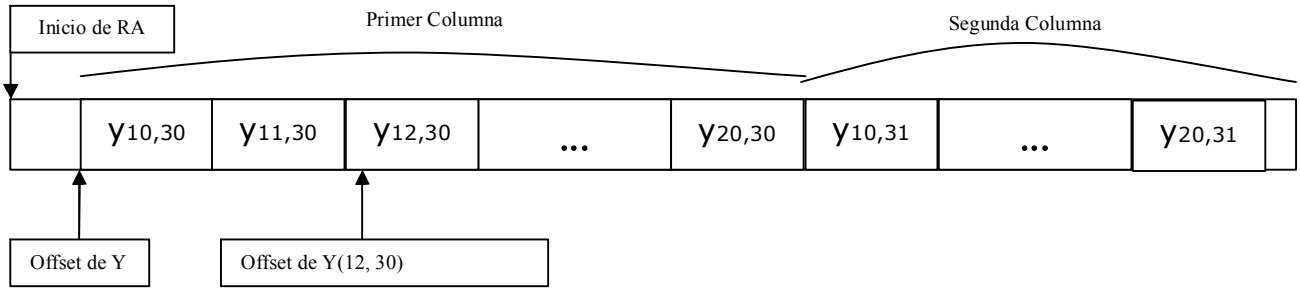
Donde:
Dv: es el offset donde comienzan los valores del array dentro del registro de activación.
Dv(i): es el offset del elemento que queremos calcular.
i: es la variable que el programador utilizó como índice de acceso.
10: es el límite inferior en la declaración del array.
TE: es el tamaño del elemento en posiciones de memoria. Por ej, si es int puede ser 2 bytes, float 4 bytes, etc.

Todos los elementos de la fórmula son conocidos en tiempo de compilación ya que el tipo es fijo, como asi también los límites del vector. Es por eso que el compilador genera las instrucciones embebidas en el código para obtener el offset Dv(i) cada vez que el programador hace referencia a un elemento del array. En el registro de activación lo único que hay guardado son los valores de los elementos del array.

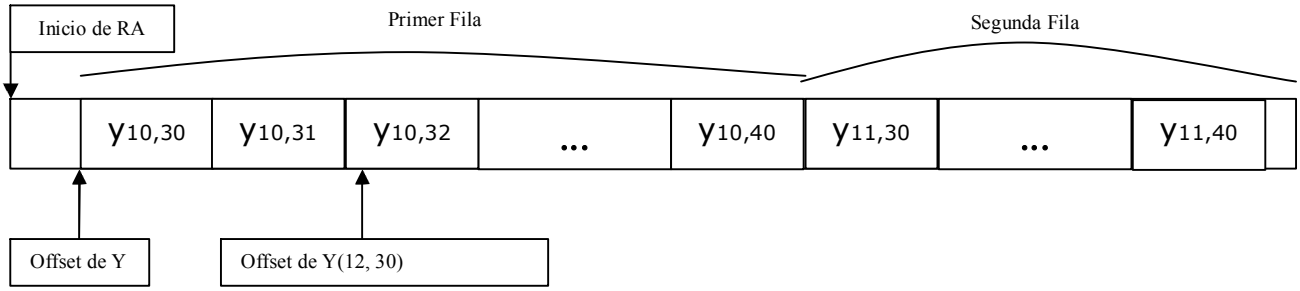
Otra variable semiestática es una matriz. Por ejemplo: int y(10 to 20, 30 to 40)

Hay dos maneras de guardar una matriz en memoria dentro del registro de activación. El lenguaje debe estipular una de las formas y guardar todas sus matrices de esa forma. En ambas se guarda de manera lineal y contigua. Las formas son:

1. Almacenamiento por columnas: Se almacena primero la línea de elementos que van desde el primer elemento hasta el último de la primer columna, luego se almacena desde el primer elemento hasta el último de la segunda columna, y así sucesivamente.



2. Almacenamiento por filas: Se almacena primero la línea de elementos que van desde el primer elemento hasta el último de la primer fila, luego se almacena desde el primer elemento hasta el último de la segunda fila, y así sucesivamente.



En estos casos, ocurre lo mismo que en los vectores, los límites son fijos y los tipos de elemento que contiene también, por lo tanto el compilador embebe una fórmula de acceso. Esta fórmula depende de cómo se almacene la matriz.

Para almacenamiento por columna, es la siguiente:

```
Dy(i, j) = Dy + [ [ (j – 30) *  (20 – 10 + 1) ] + (i – 10) ] * TE
```

Para almacenamiento por fila, es la siguiente:

```
Dy(i, j) = Dy + [ [ (i – 10) *  (40 – 30 + 1) ] + (j – 30) ] * TE
```

Donde:
Dy: es el offset donde comienzan los valores de la matriz dentro del registro de activación.
Dy(i, j): es el offset del elemento que queremos calcular.
i: es la variable de fila que el programador utilizó como índice de acceso.
j: es la variable de columna que el programador utilizó como índice de acceso.
10: es el límite inferior de fila en la declaración de la matriz.
20: es el límite superior de fila en la declaración de la matriz.
30: es el límite inferior de columna en la declaración de la matriz.
40: es el límite superior de columna en la declaración de la matriz.
TE: es el tamaño del elemento en posiciones de memoria. Por ej, si es int puede ser 2 bytes, float 4 bytes, etc.

Por ejemplo, si queremos acceder al elemento y(12, 30) con almacenamiento por columnas, la ecuación nos da lo siguiente:

```
Dy( i, j ) = Dy + [ [ (j  – 30) *  (20 – 10 + 1) ] + (i  – 10) ] * TE
Dy(12, 30) = Dy + [ [ (30 – 30) *  (20 – 10 + 1) ] + (12 – 10) ] * TE
Dy(12, 30) = Dy + [ [ (   0   ) *  (     11    ) ] + (   2   ) ] * TE
Dy(12, 30) = Dy + [ 2 ] * TE
```

Otro ejemplo, si queremos acceder al elemento y(12, 30) con almacenamiento por filas, la ecuación nos da lo siguiente:

```
Dy( i, j ) = Dy + [ [ (i  – 10) *  (40 – 30 + 1) ] + (j  – 30) ] * TE
Dy(12, 30) = Dy + [ [ (12 – 10) *  (40 – 30 + 1) ] + (30 – 30) ] * TE
Dy(12, 30) = Dy + [ [ (   2   ) *  (     11    ) ] + (   0   ) ] * TE
Dy(12, 30) = Dy + [ 22 ] * TE
```

En cada caso, si multiplicamos el resultado por el tamaño en posiciones de memoria del elemento y a eso lo sumamos a la dirección inicial de la matriz encontramos el elemento.

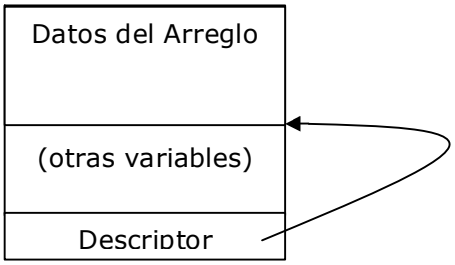
Existen lenguajes que verifican que los límites que escribe el programador al acceder a un elemento se encuentren en el rango con los que fue declarado la matriz. Por ejemplo, un acceso de y(5, 8) no sería válido en lenguajes que controlan el límite como PASCAL y ADA, sin embargo sería totalmente válido en lenguajes como C. Ese control de límites se realiza mediante instrucciones embebidas en el código:

```
If (i < limiteInferior1)
   Error;
If (i > limiteSuperior1)
   Error;
If (j < limiteInferior2)
   Error;
If (j > limiteSuperior2)
   Error;
```

Observe que los límites del matriz no se encuentran almacenados en memoria, sino que están embebidos en el código como constantes. Esto también ocurre para los vectores.

- **Semidinámicas:** son de tamaño y lugar variable en distintas ejecuciones. Solo los arreglos con límites variables caen dentro de esta categoría. Una vez que se crea el registro de activación no cambia, por ejemplo, un array declarado como x[2..**n**, 3..**m**] donde n y m son variables globales que ya tienen un valor antes de que la unidad que contiene esta declaración sea llamada. Entonces, en diferentes ejecuciones n y m tendrán distintos valores y por consecuencia el arreglo tendrá distinto tamaño, pero una vez creada este tamaño no cambia. El arreglo es dimensionado en el momento de crear el registro de activación.

  Este tipo de estructuras tienen un descriptor, que contiene los valores de los límites ya que deben estar en memoria porque al no ser constantes no se pueden embeber en el código (no se conocen sus valores al momento de la compilación). Estos límites se guardan en un descriptor (junto al tamaño de la estructura y un puntero) del cual se puede decir que se comporta como una variable semiestática. Este puntero indica el lugar donde se encuentran los datos del arreglo. Estos datos se ubican al final del registro de activación ya que al ser de tamaño variable, primero deben ubicarse las variables semiestáticas por tener un offset fijo relativo al comienzo del R.A. Observe el gráfico a continuación:



  Para compilar un acceso al arreglo, se maneja como con los arreglos semiestáticos(embebiendo la fórmula) pero en vez de dejar los valores de los límites como constantes, utiliza los valores almacenados en el descriptor.

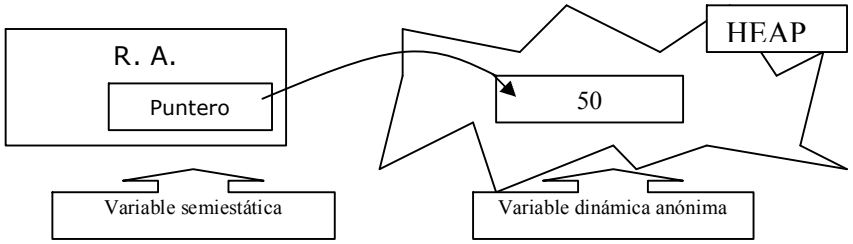
  Lenguajes como ADA tienen este tipo de variables, pero no es el caso del lenguaje C.

- **Dinámicas:** el lugar es variable y el tamaño cambia en cualquier momento luego de tener asignado espacio de almacenamiento para la variable en cuestión. El contenido de valor de estas variables no están en el registro de activación, en cambio lo que si está es un puntero que apunta a un sector de memoria llamado HEAP donde se almacenan este tipo de estructuras. Se dividen en dos tipos:

  - **Anónimas:** En C, son declaraciones como la siguiente:

    ```
    int *puntero;
    puntero = malloc(50);
    ```

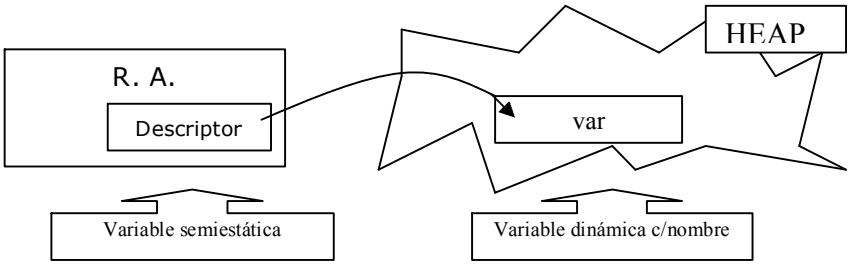
    La variable "puntero" es semiestática, se encuentra en el registro de activación y como se sabe, es un puntero a una estructura de tipo de dato int. Cuando se realiza el malloc, el compilador reserva 50 posiciones en el HEAP y hace que la variable "puntero" apunte a esta estructura (que es la variable anónima). Es el usuario el que se encarga del HEAP, pide y devuelve memoria de forma explicita como vimos en el código de ejemplo.



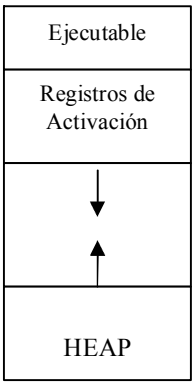
- **Con nombre:** En Algol68, son cosas como esta:

```
Flex var(1 : 0) of int
var = (1 2 3 4)
var = (1 2)
var = (1 2 3)
```

En este código se declara un arreglo de enteros y en cada asignación se le da distinta cantidad de valores (en cada una se redefine el tamaño del vector). En el registro de activación se encuentra un descriptor que entre otras cosas tiene un puntero al HEAP donde se encuentran los valores del arreglo. La diferencia con el anterior es que esta colección de valores tiene nombre y se adquiere memoria en forma implícita cuando se lo manipula.



Recordemos que el HEAP se encuentra en el mismo bloque que los registros de activación.



En lenguaje C, solo existen variables semiestáticas y dinámicas anónimas. En ADA solo existen variables semiestáticas, dinámicas anónimas y variables semidinámicas.

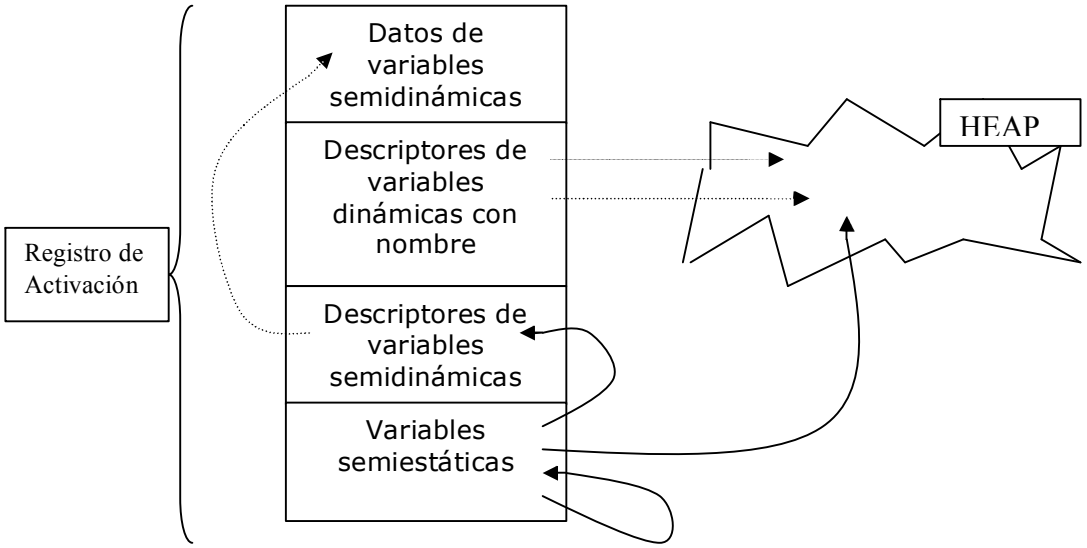
- **Superdinámicas:** Son las variables dinámicas de los lenguajes dinámicos. Obviamente cambian de tipo durante la ejecución. En los lenguajes dinámicos no existe el registro de activación, sino una tabla de símbolos y estas variables están incluidas en él por medio de un descriptor, que entre otras cosas tiene el nombre de la variable con la que se referencia en el código y un puntero al HEAP donde se encuentra el valor de la misma.

  El error de querer operar variables de distinto tipo solo ocurre en ejecución porque los tipo de valores que tienen las variables dependen de la ejecución.

  Vale aclarar que todos los lenguajes generan una tabla de símbolos en tiempo de compilación para poder generar el código, pero los únicos lenguajes que tienen tabla de símbolo en memoria mientras ejecutan son los dinámicos.

Problemas con Punteros

Como vimos, dentro del registro de activación puede haber punteros hacia otras estructuras como pueden ser a variables semiestáticas, semidinámicas, descriptores que apunten a estructuras en el HEAP, etc. Alguno de estos punteros son generados por el compilador sin que el programador sepa de su existencia, pero otros punteros son creados por el programador y es ahí donde se presentan los inconvenientes. A continuación diagramamos lo anteriormente expuesto:



Las flechas punteadas son punteros que el compilador construye y que el programador no puede ver. Con este tipo de punteros no surgen problemas ya que son administrados por el programa de forma transparente para el programador.

Las flechas con línea representan punteros administrados por el programador. Es el caso de:

- Punteros que apuntan a otra variable semiestática. Ej:

```
int a = 5;
int *p;
p = a;
```

- Variables dinámicas anónimas que tienen un puntero referenciando a un bloque de datos en el HEAP. Ej:

```
int *puntero;
puntero = malloc(50);
```

- Punteros que apuntan a variables semidinámicas.

El motivo por el cual se producen problemas con este grupo de punteros es porque el programador puede hacer una mala administración de los mismos. Veamos los problemas que pueden surgir:

- Conflicto de Tipos

  En general, el puntero puede apuntar a cualquier tipo de variable salvo que el lenguaje no lo permita. El caso clásico de este conflicto se dá en el lenguaje PL/I. Existe un tipo de dato del puntero que se le otorga al declararlo. Luego a ese puntero le puedo dar una dirección de una variable entera o de una con coma flotante. Mas adelante en determinado punto del programa no puedo determinar si ese puntero apunta a que tipo de dato porque esto depende del flujo de ejecución del programa. El código de ejemplo:

```
Declare P Pointer;
Declare A Fixed Base;
```

```
Declare B Float Base;

P := Addr(A);
P := Addr(B);
```

La mayoría de los lenguajes exige que se defina el tipo de dato al que apunta el puntero y lo controla en tiempo de compilación.

- Punteros Colgados

En este caso el programador manipula los punteros de tal forma que luego de tomar un bloque de datos, lo libera dejando un puntero señalando la zona que fue liberada. Ej en C:

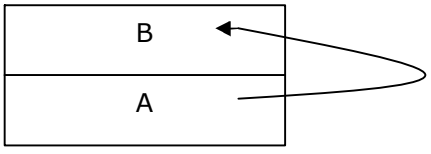
```
p = malloc(100);
q = p;
free(p);
```

En este ejemplo, se reserve un bloque de 100 bytes, luego se asigna el puntero q para que apunte a ese bloque y finalmente se libera el bloque dejando al puntero q apuntando a la zona donde ya no está.

Otro caso típico de este problema se debe al tiempo de vida de los punteros. Si tengo un puntero global y los datos son locales, cuando se termina la unidad estos datos ya no serán válidos, sin embargo el puntero seguirá apuntando a esa zona. Ej en ALGOL:

```
Proc Z1
       int x;
       ref int px;

       Proc Z2
              int y;
              ref int py;

              Px := x;  // mismo ámbito (globales): ok

              Py := x;  // al retornar, el puntero desaparece pero mantengo los datos: ok

              Px := y;  // al retornar, pierdo los datos y el puntero queda colgado: mal

              Py := y;  // mismo ámbito (locales): ok

              ...
       ...
...
```

De los cuatro casos que se explica en el código, el tercero sufre de problemas de puntero colgado ya que al terminar la ejecución de la unidad Z2, la variable "y" deja de ser direccionable pero el puntero sigue apuntando a la zona de memoria donde se encontraba. En ALGOL ese código daría error de compilación porque controla que en asignaciones de punteros que el que está a la izquierda tenga un menor o igual alcance que el que está a la derecha. Pero en otros lenguajes, esto no se controla, permitiendo el problema de punteros colgados. Es fácil esquematizar los punteros susceptibles de tener este problema, son los que apuntan a un dato que se encuentra en un registro de activación superior.



- Garbage

Esto ocurre cuando hay un bloque de datos en el HEAP del cual se perdió su puntero y por lo tanto no hay forma de acceder a él (por lo general ocurren con variables dinámicas anónimas). Ejemplo en C:

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

Primero reservé 100 bytes apuntados por "p", luego reservé otros 200 apuntado por "q" y finalmente hice que "p" apunte a los 200 bytes reservados en la segunda línea, dejándome inaccesible los 100 reservados en la primera. No tengo otro puntero apuntando al bloque perdido por lo tanto es inaccesible.

Lenguajes de Programación y Sistemas Operativos

Cada uno de estos elementos no es independiente del otro. Uno afecta al otro y viceversa. Vamos a estudiar cada tipo de Sistema Operativo y como funcionan la ejecución de los lenguajes en cada caso.

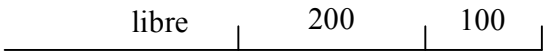
Sistema Operativo con Particiones Fijas

En estos Sistemas el programa corre en una partición fija de memoria que no varia a lo largo del tiempo. El lenguaje pone los elementos en memoria en el siguiente orden:

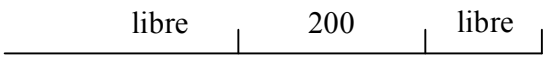
- Ejecutable
- Variables estáticas
- La pila de registros de activación
- El HEAP

Como sabemos, la pila es compacta, crece con cada llamado a una unidad y decrece cuando se retorna del llamado. En cambio el Heap es un bloque de datos desordenado, que contiene huecos libres y que crece de manera mas rápida que la pila. Ejemplo de porque ocurre esto:

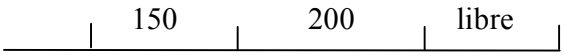
```
p = malloc(100);
q = malloc(200);
```



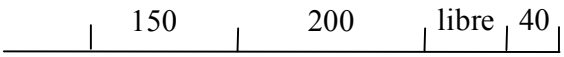

```
free(p);
```




```
p = malloc(150);
```

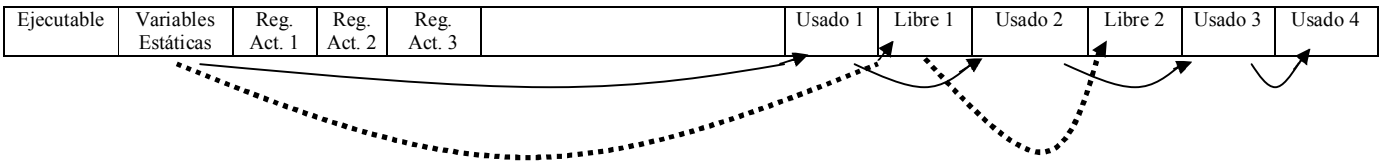



```
z = malloc(40);
```



En el ejemplo vemos como de tanto pedir y devolver memoria de distinto tamaño quedan huecos en el medio. Existen muchas políticas de asignación de memoria para elementos en el Heap pero no entraremos en detalle (en el ejemplo utilizamos First Fit, podría haber sido una Best Fit o cualquier otra). El Heap se va acercando a la zona donde se encuentra la pila y al mismo tiempo desperdiciando memoria.

Esquema de memoria para lenguajes tipo Algol y Sistemas Operativos con particiones fijas:



Aquí se representa la partición fija de la memoria donde está contenido el programa. Sobre la izquierda se encuentra el ejecutable, las variables estáticas y la pila de registros de activación, a la derecha se ubica el Heap. Existen dos estructuras que son utilizadas para mantener los espacios usados y los libres del Heap.

Lista de usados: Dentro de las variables estáticas se encuentra un puntero al primer bloque de usados del Heap. Luego, dentro de cada bloque de usado hay una lista enlazada donde cada uno apunta al siguiente dentro de la cadena(en el gráfico está representado por las flechas sólidas).

Lista de libres: Dentro de las variables estáticas se encuentra un puntero al primer bloque de libres del Heap. Luego, dentro de cada bloque de libre hay una lista enlazada donde cada uno apunta al siguiente dentro de la cadena(en el gráfico está representado por las flechas punteadas).

Al momento de reservar un bloque para usarlo, el lenguaje reserva M + N cantidad de bytes, donde M es la cantidad solicitada por el programador y N es la cantidad de bytes que ocupa un puntero, ya que en el bloque debe contenerse el puntero al siguiente usado.

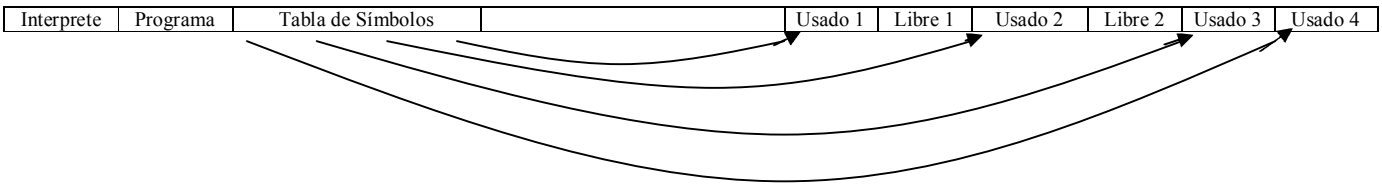
El malloc recorre la cadena de libres y le asigna un bloque que no esté siendo usado para ocupar. Va navegando por la lista desde el fondo hacia la zona de la pila en busca de la cantidad de bytes libres solicitada (en el caso de first fit).

El free pasa un bloque usado a la cadena de libres y recompone la lista enlazada. Dos libres contiguos se juntan y quedan agrupados.

Estos punteros son totalmente transparentes para el programador, los administra el lenguaje dentro de las llamadas de pedido y liberado de memoria.

El verdadero concepto de Garbage (basura) consiste en que alguno de los bloques de usados que vimos en el ejemplo no tiene un puntero que lo referencie dentro de la pila de registros de activación y a su vez (obviamente) está contenido dentro de la cadena de usados. En ese caso, estamos en presencia de fragmentación y garbage.

Esquema de memoria para lenguajes dinámicos y Sistemas Operativos con particiones fijas:



En estos lenguajes no existen las cadenas de libres y usados (no existe malloc y free tampoco). Lo que existe es una tabla de símbolos que contiene la referencia hacia el sector de Heap donde se encuentra los datos del bloque propiamente dicho. Los movimientos en el Heap tienen que ver mas con cambios de tipos (por ejemplo, que una variable pase a ser de un nuevo tipo mas grande que el anterior).

El programador no ve los punteros, solo ve las variables. Cada bloque usado tiene su puntero desde la tabla de símbolos, la relación es uno a uno. Con todo esto se deduce que no existe el Garbage como en el caso anterior, solo hay fragmentación.

Por lo general, estos tipos de lenguajes implementan un Garbage Collector que se encarga solo de compactar la memoria para eliminar bloques libres en medio de bloques usados. Esto es transparente para el programador.

Garbage Collector en Lenguajes Dinámicos

A continuación se explica el algoritmo que ejecuta:

- El interprete revisa la tabla de símbolos mirando los punteros y elige el que apunte a la dirección mas alta de memoria;
- Se pregunta si está al fondo de la partición fija;
    - Si lo está, simplemente la marca como “ya revisada”;
    - Si no lo está, corre al fondo el bloque de datos, obviamente sin pisar otros bloques que se encontraran mas atrás. La marca como “ya revisada”;
- Vuelve a realizar la operación solo que ignorando los bloques ya revisados y hasta que no quede ninguno sin revisar.

Este algoritmo se ejecuta en el momento especificado por el diseñador del lenguaje según diversos criterios que este pueda adoptar. Para el programador resulta impredecible. Existen lenguajes que permiten ejecutar el proceso con un llamado a una función.

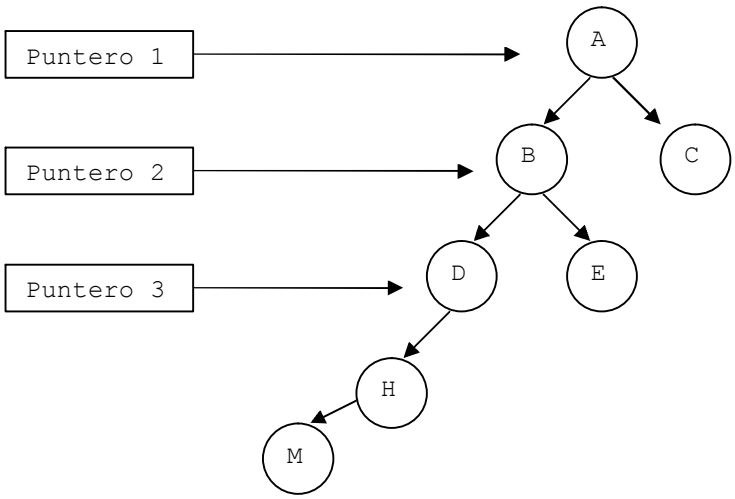
Garbage Collector en Lenguajes Tipo Algol

A continuación se explica el algoritmo que ejecuta para la eliminación de garbage:

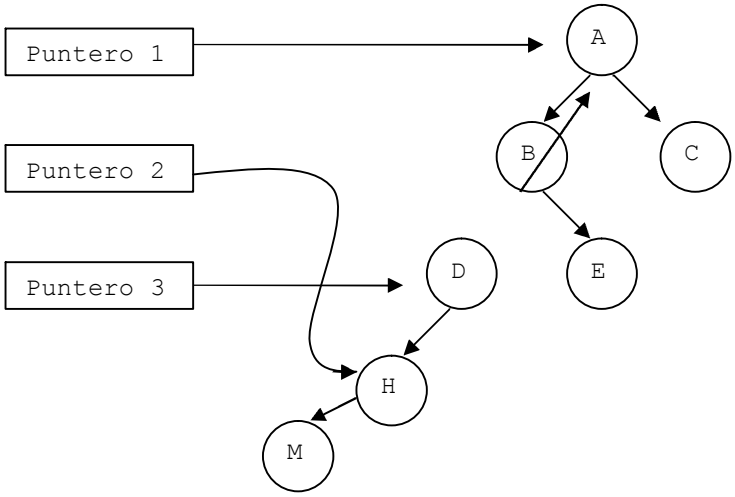
- Recorre la lista de usados y a cada uno le coloca una marca.
- Luego recorre todos los punteros que se encuentren en la pila de registros de activación y para cada bloque apuntado borra la marca que puso en el primer paso. También se fija en aquellos punteros que se encuentran contenidos en cada uno de los bloques a los que le borra la marca, con lo cual, si un bloque usado no tiene puntero en la pila pero si tiene puntero en otro bloque de usados queda incluido en este paso.
- Recorre la cadena de usados liberando los bloques que conservan la marca.

Mientras que el primer y el tercer paso son sencillos, el segundo es muy complicado. Obliga al lenguaje a tener control de todos los punteros que maneja el usuario.

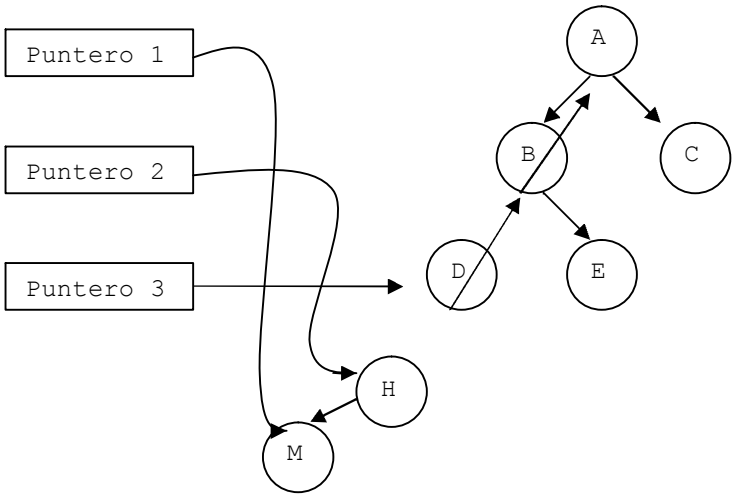
Otra complicación es que en el segundo paso, el algoritmo tiene que recorrer los bloques y los punteros de dichos bloques que son generados por el usuario. Estas estructuras pueden ser árboles, listas, grafos, etc. Por lo general, estos tipos de algoritmos están asociados con llamadas recursivas y eso necesita mucha memoria para ejecutar. Justamente, el garbage collector se ejecuta cuando falta memoria con lo cual es un problema. Sin embargo, se diseño un algoritmo que realiza esta tarea y no necesita ser recursivo. El algoritmo tiene tres punteros y con ellos va avanzando y retrocediendo dentro de la estructura. En primera instancia hace que los tres punteros apunten uno a cada nodo como figura a continuación.



Luego, avanza con el puntero 2 al nodo H pero antes doy vuelta el puntero que tiene B para que apunte a A. Esto es porque sino luego no va a tener como volver a A una vez que termina la recorrida de esta rama.



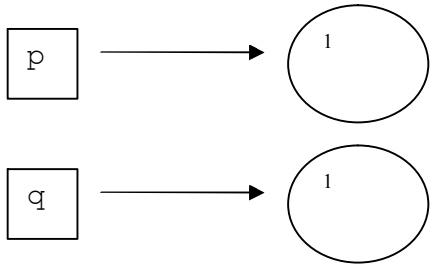
Hace lo mismo con el puntero 1.



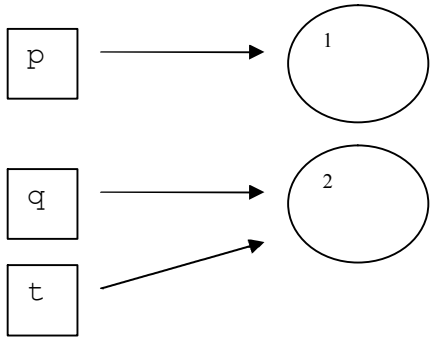
Y de esta forma va recorriendo cada nodo y realizando las marcas. El motivo por el cual da vuelta los punteros de la misma estructura es para poder volver dando pasos hacia atrás y no necesitar muchos punteros, es decir, aprovecha los punteros que ya tiene la estructura. Al ir volviendo restituye los punteros de la estructura original.

Además de eliminar el garbage, también se debe compactar el Heap. Para realizar esto, el GC debe buscar cada bloque usado y por cada uno tiene que realizar el algoritmo que se explicó anteriormente en busca de punteros que apunten a dicho bloque para poder refrescarlos. Esto es de un costo computacional altísimo aunque puede implementarse.

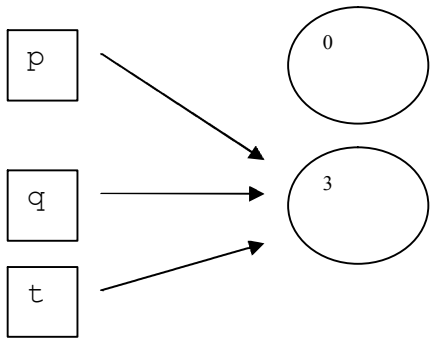
Por lo general, no existen implementaciones de garbage collector en lenguajes de tipo Algol debido a los inconvenientes que fuimos describiendo. En vez de eso, algunos lenguajes utilizan una técnica conocida como "contador de referencia". Esta técnica consiste en asignarle a cada bloque usado un valor que indica la cantidad de referencias que apuntan a dicho bloque. Si se le agrega otra referencia, el contador pasa a ser 2, si luego se le quita una referencia vuelve a 1 y cuando el bloque ya no tenga referencia se queda en 0 y es eliminado de la lista de usados. Por ejemplo, tenemos dos punteros apuntando a un bloque cada uno:



Luego se ejecuta “t = q”:

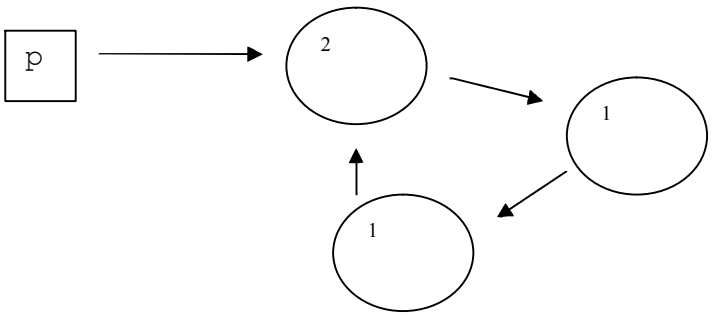


Luego “p = q”:

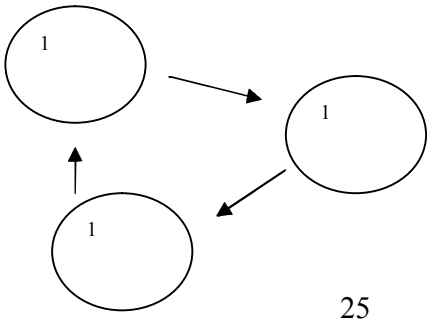


El bloque de arriba es descartado ya que su contador de referencias está en cero y la memoria es liberada.

Este algoritmo es sencillo, rápido y eficiente pero tiene un problema y es que no funciona bien con listas circulares. Supongamos lo siguiente:



Observe que los elementos tienen un puntero dentro de su estructura que apunta a otro de los elementos de la lista. A su vez, tenemos un puntero “p” que apunta a uno de los elementos y por el cual la estructura es aun accesible para el programador. Si luego eliminamos el puntero “p”, la estructura nos queda de la siguiente manera:



En este momento la estructura ya no es accesible para el programa que se está ejecutando, pero sin embargo los elementos no se eliminan de la memoria porque cada uno tiene una referencia apuntándole. El algoritmo falla cuando hay grafos. Esto no pasa para estructura tipo árbol ya que la raíz es solo apuntada por el puntero de usuario y si este se elimina, la raíz queda en cero y también se elimina. Esto ocasiona el desalojo en cascada de los elementos restantes.

Sistema Operativo con Segmentos

En este caso, el sistema operativo le entrega al programa un segmento de memoria el cual el programa no puede salir de ahí. En el caso de los lenguajes estáticos, no hay problemas ya que al conocerse cuanta memoria se necesita se le da justo la que necesita y no hay desperdicios. Para lenguajes de Tipo Algol y Dinámicos no se sabe cuanta memoria necesitará un programa de antemano, entonces el Sistema Operativo deja que lo defina el mismo programa. Por ejemplo, los EXEs de Windows tienen una cabecera en donde se incluye entre otras cosas el tamaño de memoria necesitado para ejecutar el programa. Pero el compilador tampoco puede predecir cuanto espacio va a necesitar para ejecutar, asi que necesita que el programador se lo indique. Los compiladores tienen una opción donde se le puede indicar cuanta memoria pedir al Sistema Operativo cuando se vaya a ejecutar. Si esa opción no es utilizada, siempre existe un valor por defecto que el compilador utiliza y que es un monto bastante generoso, lo que implica que sigue habiendo perdida de memoria (fragmentación interna).

Existen Sistemas Operativos que entregan diferentes tipos de segmentos a los programas (segmentos de código, de datos y de pila). En ese caso, le entrega a cada programa un segmento para el código, otro para las variables estáticas, otro para la pila y otro para el Heap. En estos casos desaparece el enfrentamiento de espacio entre la pila y el heap, pero ahora estos dos van a luchar contra el tope del segmento (permanece la misma problemática) y sigue existiendo la fragmentación interna.

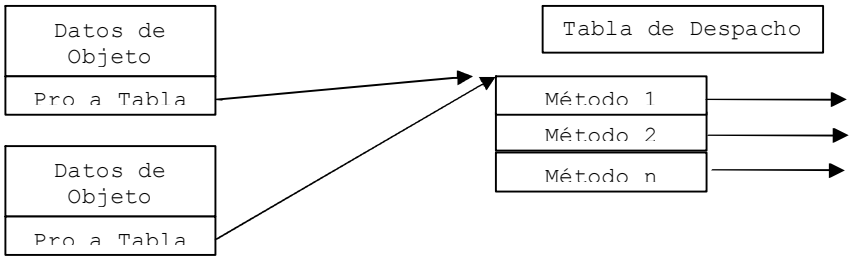
Sistema Operativo con Paginación y Memoria Virtual

En este caso, el Sistema Operativo entrega segmentos fijos muchísimo mas grandes al programa ya que estos son paginados y pueden almacenarse en el disco si no se utilizan. Los problemas son los mismos pero al tener mas espacio difícilmente la pila y el heap se choquen. La fragmentación interna podría verse como mucho mas grande pero el caso es que aquellas páginas que no sean accedidas por el programa van a estar en memoria virtual (disco rígido) con lo cual no se desperdicia la memoria principal. Lo mismo ocurre con las páginas donde haya garbage, no son accedidas y permanecen en el disco. El programador puede cometer muchas equivocaciones, generando garbage pero no es problema ya que estos están mitigados.

Nota sobre Objetos

Hasta ahora no hablamos nada de este tema. Los Objetos tienen la mismas propiedades que las variables en cuanto a la ubicación y el acceso a los mismos para lenguaje de Tipo Algol. Para lenguajes Dinámicos, se encuentran en la tabla de símbolos. La diferencia está en que mientras una variable solo tiene datos, el objeto tiene datos y punteros a los métodos que implementa:

Los objetos de lenguajes mas modernos tienen una tabla de despacho común para instancias de la misma clase donde se encuentran los punteros a los métodos:



Concurrencia y Hebras de Ejecución

Sabemos que un Sistema Operativo puede manejar concurrencia, un ejemplo de esto es UNIX. También sabemos que existen lenguajes que ejecutan tareas concurrentes, por ejemplo ADA. Estas dos cuestiones son bien diferentes, por un lado está la concurrencia que maneja el Sistema Operativo (tiene la capacidad de administrar ejecuciones paralelas de diversos procesos) y la que maneja el Lenguaje (posee estructuras e instrucciones que puede usar el programador para administrar de forma paralela diversas tareas de un mismo programa). Estas cuestiones se presentan y pueden coordinar entre las dos. Veamos cada uno de los ambientes:

- Sistema Operativo sin concurrencia / Lenguaje sin concurrencia: Obviamente, no existe la concurrencia en este sistema.

- Sistema Operativo con concurrencia / Lenguaje sin concurrencia: Existe la concurrencia administrada por el S. O. El lenguaje cree que él solo está corriendo en el procesador mientras que el S. O. conmuta entre procesos independientes.

- Sistema Operativo sin concurrencia / Lenguaje con concurrencia: Existe la concurrencia administrada por el lenguaje. Este es el que implementa los métodos de sincronización y lo que se conmuta es entre procedimientos del mismo programa.

- Sistema Operativo con concurrencia / Lenguaje con concurrencia: Dentro de esta categoría puede darse dos situaciones distintas:

    - El Sistema Operativo y el Lenguaje no se conocen: Esto significa que uno no sabe que el otro es concurrente. Existen dos colas de CPU, una la maneja el S. O. (donde se conmuta por proceso) y la otra el Lenguaje (donde se conmuta por procedimiento dentro del programa). Cuando un programa es seleccionado de la cola de listos para ejecutar, este es el que indica que procedimiento es el que se ejecutará en ese ciclo. También se manejan prioridades independientes (el S.O. no reconoce prioridades entre los procedimientos de un proceso) y de esta forma no se pueden evaluar en conjunto.

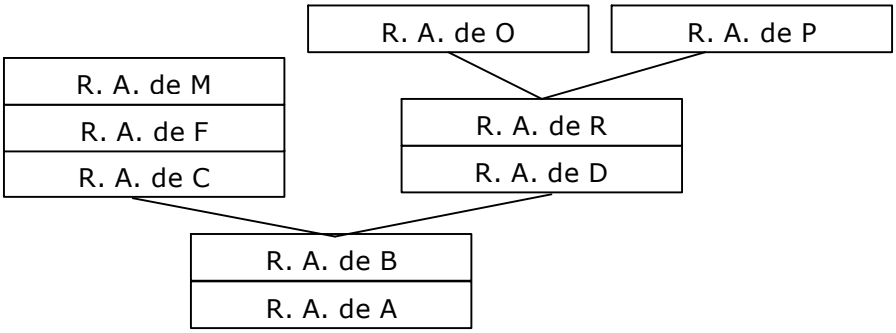
    - El Sistema Operativo y el Lenguaje se conocen: El lenguaje descansa en el S. O. para la implementación de la cola de listos. El S.O. conmuta entre procedimientos (hebras) del mismo

proceso analizando sus prioridades. En este punto es donde el S. O. empieza a conocer acerca de las características de los procesos que ejecuta.

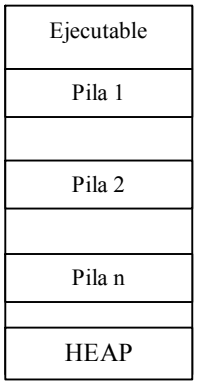
Teniendo concurrencia, un programa deja de ser lineal, con lo cual una secuencia de llamados como las que veíamos antes puede ser de la siguiente manera:

→ C → F → M
A → B
→ D → R → O
→ P

Con esta situación, ya no se puede utilizar una pila como la estábamos utilizando antes porque pierde su carácter de LIFO. Si armamos la pila de llamadas como lo hacíamos antes, seria algo como lo siguiente:



Para realizar esto, el compilador almacena en un sector distinto y alejado de la memoria a cada registro de activación. Con esto cada pila puede chocar con cualquier otra e incluso con el Heap. Pero recordemos que en S. O. con memoria virtual teníamos mucho espacio en el segmento con lo cual no debemos preocuparnos.



Para resolver donde poner las pilas, el S. O. solicita que el programa se lo indique y a su vez este al programador. Si no es indicado, se ubica cada pila en la mitad del segmento que le quede disponible. Es decir, la primer pila adicional va a ir en la mitad del segmento, la segunda en la mitad de la mitad, etc.

Pasaje de Parámetros

Se sabe que cuando una unidad invoca a otra, le puede enviar parámetros de entrada y recibir una respuesta luego de la ejecución. Para esto, la unidad que invoca se le llama "unidad llamadora" y la que es invocada se le llama "unidad llamada". La unidad llamadora tiene los parámetros "reales" a la unidad llamada, la cual define los parámetros "formales".

Clasificación según la sintaxis:

- Asociación Posicional: Cada parámetro real se asocia con uno formal según la posición que ocupan. Dentro de esta categoría se dividen en:
    - Sin Faltantes: Tiene que ser uno a uno, no hay faltantes. En cada llamado existe la misma cantidad de parámetros reales que de formales.
    - Con Faltantes: Un llamado a una unidad puede no tener especificado todos los parámetros que se definen en la declaración formal Dentro de estos se clasifican en:
        - Faltantes al Final: El lenguaje permite que la llamada a la función no tenga la misma cantidad de parámetros en en la definición formal pero los que falten son los que se encuentran al final. Ejemplo:

          FuncionA (a, b, c) → FuncionA (int x, int y, int z, int w)

          La llamada que se encuentra a la izquierda tiene tres parámetros reales mientras que la declaración de la función tiene cuatro formales. El valor que no se suministra es el de "w" y el lenguaje debe implementar algún método para definir el valor de ese parámetro cuando no es explicitado.
        - Faltantes en Cualquier Lugar: El lenguaje ofrece una forma de indicar que parámetros son los que se le pasa el valor y cuales no. Ejemplo:

          FuncionA (_ , _ , 3) → FuncionA (int x, int y, int z)

- Asociación Explicita: En la llamada se indica explícitamente que valor real se asigna a que parámetro formal. Ejemplo en ADA:

  FuncionA (x as 3, y as 4) → FuncionA (x, y);

- Asociación Anónima: La unidad recibe una cantidad variable de parámetros a partir de algún mecanismo implementado por el lenguaje.
  Ejemplo en C:

  int funcionX (float b, ...);

  Esta función puede ser llamada por diferentes llamados como estos:

  FuncionX(55, a);
  FuncionX(2, 3, 8, 4);

  Un ejemplo de esto es el printf.
  Ejemplo:

  printf("Valores: %d, %d", a, b);

Los parámetros pueden tener un valor por defecto si es que en la llamada no se le pasa valor.
Ejemplo:

int Funcion(int x, int y, float z = 0.5);

Clasificación según la semántica:

- Referencia: Esto consta en que el compilador se la rebusca para que cada uno de los parámetros reales sean referenciados en el cuerpo de la unidad llamada cuando los parámetros formales son utilizados. Para ser mas claros, cuando una unidad A llama a una unidad B, se crea el registro de activación de B pero no se reserva lugar para los valores de los parámetros formales, sino que cuando la unidad utilice a uno de ellos en realidad tendrá un acceso al registro de activación de A donde se encuentran los parámetros reales.

- Nombre: El compilador reemplaza los parámetros formales por los reales textualmente en la función llamada. Esto lo usa Algol y hoy dia en las Macro.

- Copia: En este caso, en el registro de activación de la unidad llamada existe el espacio dedicado a los parámetros formales. Este se clasifica en los siguientes:

    - Copia Valor: El compilador en cada invocación genera las instrucciones necesarias para copiar el valor de los parámetros reales dentro del espacio de almacenamiento de los parámetros formales en el registro de activación llamado. Esto ocurre al principio del llamado.
      Ejemplo:

      ```
      A(a, b) → B(x, y)
      ```

      El compilador crea estas instrucciones el momento antes de realizar el salto a la nueva unidad:

      ```
      x = a
      y = b
      Call ....
      ```

    - Copia Resultado: Realiza exactamente lo contrario, asignando los valores resultantes sobre los parámetros reales al finalizar la ejecución de la unidad llamada.
      Ejemplo:

      ```
      A(a, b) → B(x, y)
      ```

      El compilador crea estas instrucciones el momento antes de realizar el retorno a la unidad llamadora:

      ```
      a = x
      b = y
      Return ....
      ```

    - Copia Valor / Resultado: Realiza ambas cosas.

El lenguaje C solo tiene pasaje por Copia Valor y hay que destacar que no tiene pasaje por referencia. Esto a veces se confunde porque en este lenguaje lo que se permite es pasar una dirección de una variable por Copia Valor. Pero la dirección de esa variable se encuentra almacenada en el registro de activación de la unidad llamada.

Ejemplo de Ejecución con Diferentes Pasajes

Vamos a ejecutar mentalmente el siguiente programa escrito en un lenguaje ficticio y obtener el resultado final de cada una de las variables involucradas:

```
int elem;
int a(1..2)

procedure pasaje(int x)
       a(1) = 6;
       elem = 2;
       x = x + 3;
end pasaje

procedure Main()
       a(1) = 1;
       a(2) = 2;
       elem = 1;
       pasaje(a(elem));
end Main
```

Por Referencia: Al ejecutar el llamado a pasaje, a esta unidad se le está pasando la dirección física del elemento 1 del array. Por eso dentro de la misma, las refrencias a "a(1)" y "x" desembocan en la misma celda de memoria, con lo cual primero se le asigna 6 y luego se le suma 3 guardándola en el mismo lugar.

Resultado:

A(1)	A(2)	Elem
9	2	2

Por Nombre: El compilador genera el reemplazo textual de los parámetros quedando asi:

```
procedure pasaje(int x)
       a(1) = 6;
       elem = 2;
       a(elem) = a(elem) + 3;
end pasaje
```

Con lo cual, en la tercer instrucción utiliza la celda 2 del array ya que en medio se le modifica el valor a "elem".

Resultado:

A(1)	A(2)	Elem
6	5	2

Por Copia Valor: Simplemente la x es una variable independiente que se aloja en el registro de activación de la unidad llamada y a la que se le copia el valor pasado como parámetro.

Resultado:

A(1)	A(2)	Elem
6	2	2

Por Copia Valor/Resultado: Pasa lo mismo que en el anterior con la diferencia que al finalizar el procedimiento el valor de "x" es copiado a la dirección de la variable que se utilizó como parámetro de entrada(en este caso "a(1)").

Resultado:

A(1)	A(2)	Elem
4	2	2

Evolución del Pasaje de Parámetros

Al principio cuando se creo el compilador de Foltran quisieron que los pasajes de parámetros se hagan por referencia. El problema surge cuando a un procedimiento que se le pase una constante como parámetro este le asigne un valor a dicho parámetro. Ejemplo:

```
FuncionX (8, a)  →        FuncionX (x, y)
                                ....
                                ....
                                ....
                                x = 6
                          End FuncionX
```

En el ejemplo vemos como el parámetro real con el que se llama a la función es un 8 mientras que dentro del cuerpo de la misma se le asigna un valor al parámetro formal x. No hay donde guardar ese valor ya que el parámetro fue una constante. Esto provoca la imposibilidad de pasar constantes o expresiones por referencia.

Mas tarde se creo Algol pero este no uso pasaje por referencia. Lo que uso fue el pasaje por nombres. Pero a este método le ocurría el mismo problema. Veamos un ejemplo con expresiones:

```
FuncionX (b + 3, a)  →    FuncionX (x, y)
                                ....
                                ....
                                ....
                                x = 6
                          End FuncionX
```

El compilador reemplaza textualmente la asignación para resolver el llamado y queda asi:

```
FuncionX (x, y)
       ....
       ....
       ....
       b + 3 = 6
End FuncionX
```

No se puede realizar la asignación, sigue teniendo problemas. Por eso crearon el método por copias.

ADA es un lenguaje que maneja todos estos métodos y lo hace a través de su sintaxis:

```
FuncionX (in A, in B, out C, in out D);
```

En este caso la sintaxis dirige a la semántica.

Compatibilidad y Conversión entre Tipos de Datos

Cuando tenemos una asignación “a = b” pueden ocurrir dos cosas diferentes dependiendo del tipo de lenguaje de programación donde se ejecute:

- Lenguaje Dinámico: la variable “a” pasa a ser del mismo tipo que la variable “b” y ambas referencian al mismo dato.
- Lenguaje Tipo Algol: el tipo de dato de la variable “b” debe tener forma de transformarse al tipo de dato de la variable “a”.

De aca en mas, todo lo que digamos va a aplicar a los lenguajes de Tipo Algol. Si la sentencia anterior “a = b” compila, se dice que ambos tipos de datos son compatibles. Existen dos tipos de compatibilidad:

- Nombre: Se refiere a cuando los tipos de datos son incompatibles tan solo con tener distinto nombre. Ejemplo en ADA:

```
type reales is new float
       A: reales
       B: float
…
A + B  → Error de compilación
```

  El programa anterior no compila porque las variables son de distinto tipo a pesar que en definitiva, "reales" es en realidad un "float".

- Estructura: Se refiere cuando los tipos de datos son incompatibles solo si tienen estructuras diferentes (cantidad de bits y significado de los mismos). Ejemplo en C:

```
typedef float reales
       reales A;
       float B;
…
A + B  → ok y se ejecuta
```

  Si bien los tipos de datos parecen ser diferentes, son compatibles porque ambos tienen el mismo tamaño y cada uno de sus bits significan lo mismo(es decir, la misma estructura).

Como vimos, C tiene compatibilidad por estructura mientras que ADA por nombre. ¿Porque los diseñadores de ADA lo hicieron asi? Para poder diferenciar valores que conceptualmente sean distintos. Ej:

```
type dolar is new float
type pesos is new float
       A, B: dolar
       C, D: pesos
       E, F: float

A = C + E                  → Error de compilación
E := float(A + dólar(C))   → Realiza las transformaciones explícitamente, esto está permitido
```

Esto hace a ADA poco escribible pero muy legible.

Existen dos tipo de conversiones entre tipos:

- Implícitas: Son las que el compilador genera automáticamente cuando a una variable de un tipo se le asigna otra de un tipo compatible.
- Explícita: El programador escribe estas conversiones dentro de las expresiones.

Nota aparte: ADA tiene tipos derivados. Por ejemplo:

```
Subtype euros is float
Subtype edad is float range 0..110
```

Estos se comportan igual que un "float" porque es un subtipo, se puede sumar "float" y "edad". Si en medio de la ejecución, una variable de tipo "edad" se va de rango, se arroja un error "out of range".

<u>Conversiones Implícitas en Algol (Coerciones)</u>

- Voiding: En Algol es posible definir una variable de tipo Void (nulo). Estas variables sirven para descartar resultados de funciones. Este tipo de datos no es equivalente al puntero a Void de C.

```
void x;
x := 2;          // El valor no se almacena en ningún lado
x := Fun(2, a); // En Algol no se podía ignorar el retorno de una función salvo con void
```

- Widening: Esta es una conversión de tipos donde el valor "se ensancha". Por ejemplo, una variable int se transforma en un real.

```
real a;
int b;
a := 2;  // Las dos líneas contienen widening
a := b;
```

- Rowing: diferentes autores dicen distintas cosas a cerca de este tipo de conversión, pero vamos a explicar una de ellas. Los arreglos en Algol se llaman rows, y consta de la asignación de un valor al arreglo de manera que todos los elementos del arreglo se inicialicen con dicho valor.

```
[4:20] int z;
z := 4;       // todos los elementos de z adquieren el valor 4
```

- Uniting: Una unión en Algol es una variable que puede tener mas de un tipo. Ejemplo:

```
union (int, real) w;
```

Supongamos que los int ocupan 2 bytes y los real 4 bytes, entonces la estructura "w" ocupará al menos 5 bytes. Esto se debe a que la estructura tiene espacio para guardar el mas grande de los valores y un byte mas para indicar que tipo está guardando a cada momento. Ese campo se llama discriminante.

```
w := 4;
```

Con esta asignación la estructura tomará el valor 4 con el tipo de dato int. Esto significa que el valor ocupará 2 bytes, dejando otros 2 bytes sin uso y el discriminante indicará que se está almacenando un int.

```
w := 6.5;
```

Esto también es válido y con esta asignación la estructura tomará el valor 6.5 con el tipo de dato real. Esto significa que el valor ocupará los 4 bytes del campo de valor y el discriminante indicará que se está almacenando un real.

Con esto el lenguaje provee cierto dinamismo en tipo de variables en un lenguaje tipo Algol pero esos tipos están sujetos a los que se definió en la declaración.

Pero existe un problema cuando se utiliza una variable de este tipo a la derecha en una asignación:

```
int z;
union (int, real) w;
. . .
z := w;
```

Esto no compila porque el compilador no puede asegurar que "w" tenga el tipo int en ese momento para asignárselo a "z". Para solucionar este tema se inventó el concepto de cláusula de conformidad. Esta es una estructura de decisión que se ejecuta consultando el discriminante en la variable de unión. Ej:

```
case w in:
       when w int: z := 3 * w + h;
       when w real: m := 7.1 * w;
esac
```

Dependiendo del tipo de valor se ejecuta una de las asignaciones o cualquier otra instrucción que se ponga bajo la cláusula "When". De esta forma el compilador acepta la asignación y asi Algol asegura que las uniones son seguras.

- Desproceduring : Para ver este concepto primero debemos explicar lo siguiente. Ejemplo:

```
proc xx( . . . )  . . .
proc yy( . . . )  . . .
```

Puedo declarar un procedimiento sin indicar sus instrucciones:

```
proc tmp;
```

Entonces, puedo hacer los siguiente:

```
tmp := xx;  // El procesos tmp ahora es el mismo que xx
```

En este momento si invoco a "tmp" el proceso que se ejecuta es el declarado en "xx". También puedo hacer:

```
xx := yy;
yy := tmp;
```

Esto se implementa definiendo a todos los procedimientos como un puntero a las instrucciones. Esto se hizo asi porque siempre se buscó hacer programas genéricos. En C se hizo lo mismo pero con los punteros a funciones. Esto es la noción inicial de lo que después fue la herencia en programación orientada a objetos.

Cuando en la asignación tengo un procedimiento a la derecha y una variable numérica a la izquierda, en vez de asignar el procedimiento a otra variable, este se ejecuta y retorna un valor. La acción de ejecutar el procedimiento en vez de retornar su dirección se llama Desproceduring. Ejemplo:

```
int a;
a := xx( . . . );
```

Los procedimientos se ejecutan si tienen una variable a la izquierda de la asignación.

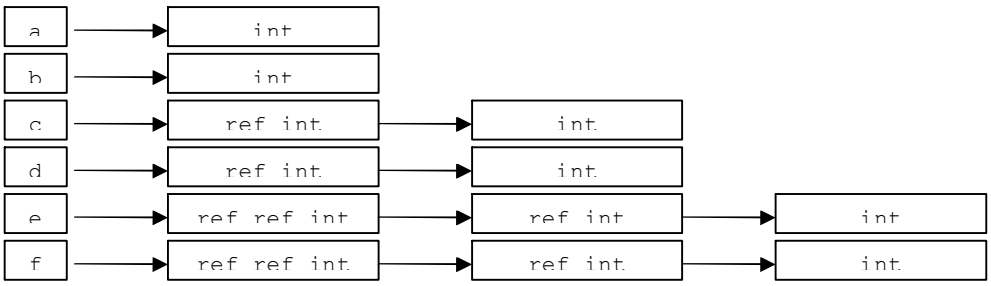
- Desreferencing: Para ver este concepto primero debemos explicar lo siguiente. Vamos a declarar las siguientes variables en Algol:

```
int a;
int b;
ref int c;
ref int d;
ref ref int e;
ref ref int f;
```

En el código anterior se declaran dos variables de tipo int, dos variables que son un puntero a un int y otras dos que son un puntero a un puntero a un int. Aca vemos el mismo código pero en lenguaje C para que se entienda mejor:

```
int a;
int b;
int *c;
int *d;
int **e;
int **f;
```

Gráficamente se puede representar las variables anteriores de la siguiente manera:



Algol define que cada variable representa la dirección donde se encuentra el dato. Como vemos, "a" y "b" apuntan directamente al dato en memoria, es decir, para referenciar su valor el programa realiza una sola búsqueda en memoria para obtenerlo. En el caso de "c" y "d", las variables representan la dirección de una celda de memoria que contiene la dirección del dato final. Para obtener dicho valor, el programa realiza dos accesos a memoria, primero para buscar el valor del puntero y con ese valor realiza un segundo acceso para obtener el dato. Las variables "e" y "f" son punteros a punteros de int, asi que realizan tres accesos a memoria para obtener el dato.

Algol tiene la particularidad de definir que en una asignación, la parte derecha debe rebuscársela para devolver el tipo de valor que está solicitando la parte izquierda. Por ejemplo, si tenemos "a := ..." lo que haya a la derecha debe devolver un int, si tenemos "c := ...." debe devolver una referencia a int y si tenemos "e := ..." debe devolver una referencia a una referencia de int.

Veamos ejemplos:

- Asignaciones que esperan tipo int:
  En estas asignaciones la variable a la izquierda es "a" de tipo int, esto significa que el valor esperado es un valor de tipo int.

  - a := b;
    Como "b" que está a la izquierda también es un int, lo que hace el compilador es buscar en memoria el valor de "b" y asignárselo a la variable "a". Se dice que esta asignación hace **1 desreferencing** porque para obtener el valor de "b" hace una búsqueda en memoria("b" es la dirección de dicho valor). Note que el acceso a memoria para guardar en valor en "a" no es tenido en cuenta.

  - a := c;
    La variable "c" es un puntero a un int. Entonces, como se le pide un valor de int lo que hace el compilador es buscar en memoria la referencia y luego con ella buscar el valor definitivo de tipo int. Esto hace dos accesos a memoria por lo tanto se dice que hay **2 desreferencing**.

  - a := e;
    De la misma forma que el anterior, se solicita un int y la única forma con "e" es realizar los tres accesos a memoria que nos llevan a encontrar dicho valor. Hace **3 desreferencing**

- Asignaciones que esperan una dirección a un tipo int:
  En estas asignaciones la variable a la izquierda es "c" de tipo ref int, esto significa que el valor esperado es una dirección que contenga un valor de tipo int.

  - c := a;
    Habíamos dicho que en Algol las variables representan las propias direcciones en memoria de sus valores. En esta asignación, como "c" espera por una dirección de un int, esa dirección ya está embebida en "a" con lo cual no necesita ir a memoria para buscar la dirección del valor. Por lo tanto **no hay desreferencing**.

- c := d;
  La variable "d" contiene un puntero a int. Para obtener su valor se debe acceder a memoria con lo cual se realiza **1 desreferencing**.

- c := e;
  La variable "e" contiene un puntero a un puntero a int. Como solo se espera un puntero a int, el compilador lo que hace es primero acceder a memoria para obtener el primer puntero de "e" el cual contiene la dirección de otro puntero y al cual accede para obtener su valor y devolverlo a "c". Son dos accesos a memoria, con lo cual hay **2 desreferencing**.

- Asignaciones que esperan una dirección de una dirección a un tipo int:
  En estas asignaciones la variable a la izquierda es "e" de tipo ref ref int, esto significa que el valor esperado es una dirección que contenga la dirección de un valor de tipo int.

  - e := a;
    Esta asignación es **imposible** y arroja error de compilación. Esto se debe a que "a" es una dirección de un int y no hay forma de devolverle un puntero a puntero de int.

  - e := c;
    En este caso, la variable "c" en el código ya es la dirección de una celda que contiene un puntero a int. Con lo cual, si le damos a "e" esa misma dirección que contiene la variable "c" encaja perfecto en lo que pide. Por lo tanto **no hay desreferencing**.

  - e := f;
    Se accede a memoria una vez para obtener el valor que contiene la dirección apuntada por la variable "f" y se lo asigna en "e". Hace **1 desreferencing**.

La idea de Algol era matar la diferencia entre la dirección de un valor y el contenido del mismo. Esto dio confusión a los programadores. A partir de Algol los lenguajes modernos tienen un desreferencing implícito. Por ejemplo, escribamos las mismas asignaciones en lenguaje C:

- Asignaciones que esperan tipo int:

  - a = b;
    1 desreferencing

  - a = *c;
    2 desreferencing.

  - a = **e;
    3 desreferencing

- Asignaciones que esperan una dirección a un tipo int:

  - c = &a;
    No hay desreferencing

  - c = d;
    1 desreferencing

  - c = *e;
    2 desreferencing.

- Asignaciones que esperan una dirección de una dirección a un tipo int:

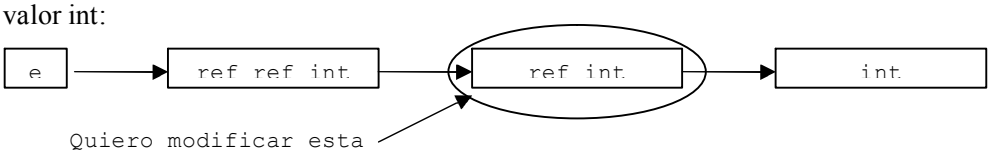
- e = &c;
  No hay desreferencing

- e = f;
  1 desreferencing

El operador & suprime el desreferencing implícito en C. Por ejemplo, en la asignación "c = &a;" si no agregamos el operador, el compilador lo que haría es buscar el valor que contiene la variable "a" en vez de darle su dirección a "c". Algo parecido ocurre con los operadores * que son necesarios incluirlos para agregar desreferencing a la asignación además del que ya se realiza implícito.

Vale aclarar que todo esto es asi porque el lenguaje C no construye automáticamente las instrucciones para que el valor devuelto en la parte derecha de la asignación encaje con la parte izquierda como pasa en Algol.

Un caso particular: Si tengo la variable "e" del ejemplo y quiero alterar la celda que apunta directamente al valor int:



Debo poner lo siguiente:

```
(ref int) e := ....
```

Esto hace que se omita un desreferecing. Esta operación es llamada CASTING en Algol (pero no es lo mismo que el casting de lenguaje C).

Control de Precisión

El resultado de ciertas expresiones de un programa pueden depender de la forma que tiene el mismo de representar los valores de coma flotante. Ejemplo:

```
float x
x := 0.1
if (x * 10.0 == 1.0)
       then print "Que bien!"
       else print "Que mal!"
```

Para saber que es lo que se imprime en pantalla al ejecutar el programa se debe saber el tipo de aritmética del lenguaje. Existen dos formas de estructurar un dato en coma flotante: IEEE y BCD. En lenguaje C, se puede compilar con la biblioteca STDLIB (utiliza IEEE) y el resultado del programa daría "que mal", mientras que si se compila con CBDLIB (utiliza BCD) muestra "que bien". En Pascal siempre se utiliza IEEE por lo tanto devuelve "Que mal". Vemos los formatos:

- IEEE: El dato tiene varios campos: signo, mantisa y exponente. En simple precisión tiene 32 bits (1 de signo, 23 de mantisa y 8 de exponente. En doble precisión son 64 bits (1 de signo, 52 de mantisa y 11 de exponente). Este formato es apto para realizar operaciones matemáticas rápidas y es compacto pero tiene un problema, al ser una representación binaria existen ciertos números que son periódicos pero en decimal no lo son. Por ejemplo, en decimal el 0.1 es exacto y en binario ese mismo número es periódico. Por eso, en el código de ejemplo, cuando se realiza la operación "x * 10.0", al tener "0.1" esto no da exacto sino un número periódico que al compararse con 1.0 al programa le resulta distinto.

- BCD: De las siglas Binary Code Decimal, este formato tiene la particularidad de guardar cada dígito decimal con cuatro bits. Por ejemplo, el valor 13.6 en BCD esta representado por la siguiente cadena de bits:

0001 0011 . 0110

Esto hace que el formato represente fielmente los números decimales ya que el 0.1 no es periódico, sino una fiel representación del mismo. La contra de esto es que los valores ocupan mas espacio de almacenamiento (simple de 32 bits y doble de 80 bits) y que las operaciones matemáticas son mucho mas lentas.

Para profundizar sobre IEEE decimos por ejemplo que "5 * 1/5" no nos dá 1 como pensamos, porque el 1/5 es periódico en binario.

Algunos lenguajes y los formatos:

- Cobol usa BCD porque necesita exactitud.
- Fortran usó un Pre IEEE.
- C usa ambas.
- Pascal usa una IEEE.
- ADA usa ambas pero dirigido por la sintaxis. Ejemplo:

  type Pesos is new float delta 0.01

  Esto indica que el float se maneja con BCD con 2 dígitos a la derecha de la coma. Es decir, el valor 12.30 y 12.31 son continuos inmediatos, no existe nada entre ellos.

  type Pesos2 is new float delta 0.001
  A: Pesos
  B: Pesos2
  A = A + Pesos(B) / B = B + Pesos2(A)

  Tengo que convertir para sumar.
  Si no pongo lo de "delta", el float se maneja en IEEE.

BNF

Backus Norm Form es un lenguaje de definición de sintaxis de lenguajes (un meta lenguaje). En 1957 Backus creo Foltran y uso BNF para crearlo. Desde ese momento los lenguajes se definen utilizando BNF.

Elementos utilizados para definir un lenguaje en BNF:

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

Entidades No Terminales: < y > 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 que vimos en el item anterior.

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, la letra a, todos los dígitos, etc.

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

Vamos a dar un ejemplo de una sentencia BNF que define un token válido:

<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. Esto nos da una idea de anidamiento entre reglas. Por ejemplo:

Veamos si la variable **num4** es válida:

La "n" es una <LETRA>, asi que por ahora tenemos:
<LETRA>um4
Una <LETRA> es una <VARIABLE> según dice la tercer regla, asi que tenemos:
<VARIABLE>um4
Si tomamos <VARIABLE>u por la cuarta regla sabemos que una <VARIABLE> es también una <VARIABLE><LETRA>, y como "u" es una <LETRA> entonces nos queda lo siguiente:
<VARIABLE>m4
Si repetimos la operación anterior pero con <VARIABLE>m, tenemos lo siguiente:
<VARIABLE>4
Y la última regla nos dice que <VARIABLE><DIGITO> también es una <VARIABLE> asi que, como 4 es un <DIGITO> nos queda la entidad no Terminal como resultado:
<VARIABLE>

Pudimos partir del texto original y verificamos que ese texto representa a la entidad <VARIABLE> dentro de nuestro lenguaje definido por las reglas anteriormente vistas. Este texto entonces es válido dentro de nuestro programa.

Veamos si la variable **8y** es válida:

El "8" es un <DIGITO> asi que tenemos:
<DIGITO>y
Como "y" es una <LETRA> tenemos:
<DIGITO><LETRA>

No hay regla que defina mas que esto, asi que no pudimos llegar a la definición de <VARIABLE> por lo tanto "8y" no es una variable válida para nuestro lenguaje.

Arbol de parsing: Es una estructura que se realiza para comprobar si una expresión esta correctamente escrita según las reglas BNF dadas, es decir, se utiliza para que dado un programa escrito en determinado lenguaje y las reglas BNF que componen su sintaxis, poder saber si el programa es válido o no.

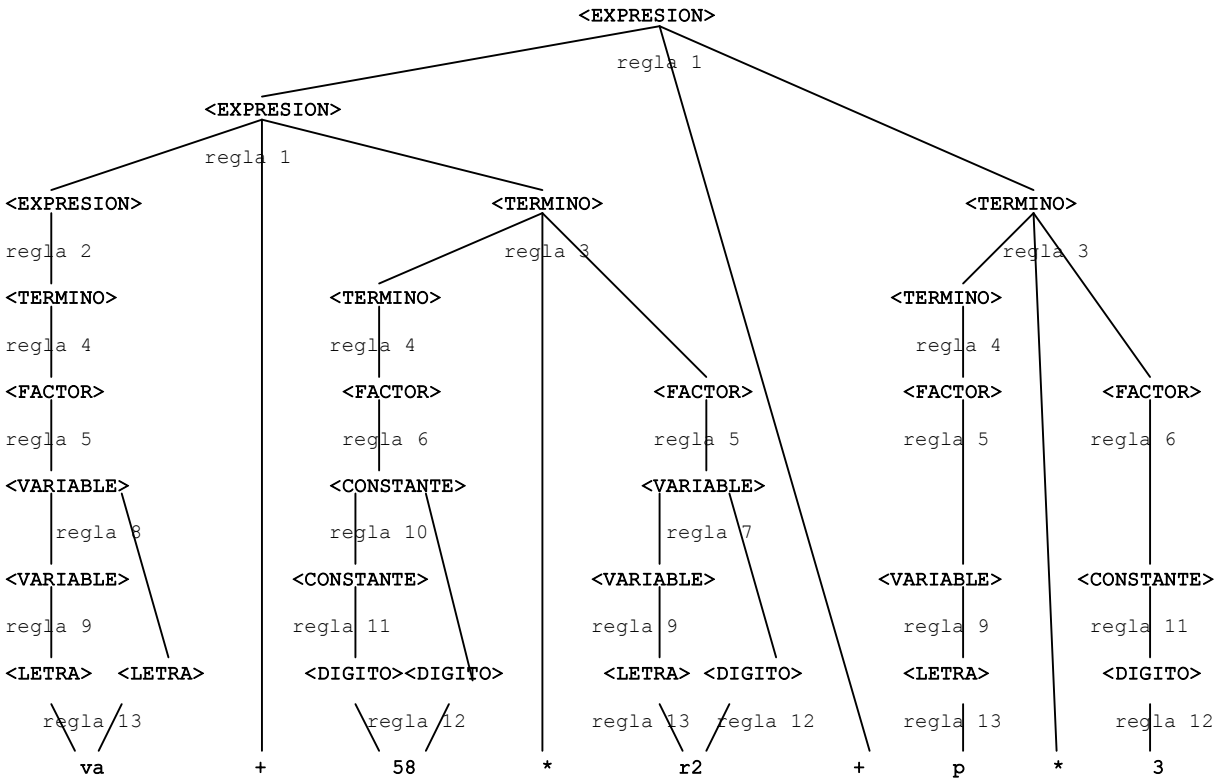
Ejemplo de árbol de parsing:

BNF:

1. <EXPRESION> → <EXPRESION> + <TERMINO>
2. <EXPRESION> → <TERMINO>
3. <TERMINO> → <TERMINO> * <FACTOR>
4. <TERMINO> → <FACTOR>

5. <FACTOR> → <VARIABLE>
6. <FACTOR> → <CONSTANTE>
7. <VARIABLE> → <VARIABLE><DIGITO>
8. <VARIABLE> → <VARIABLE><LETRA>
9. <VARIABLE> → <LETRA>
10. <CONSTANTE> → <CONSTANTE><DIGITO>
11. <CONSTATE> → <DIGITO>
12. <DIGITO> → 0 | ... | 9
13. <LETRA> → a | ... | z

Teniendo estas reglas, hacemos el árbol de parsing para verificar si la siguiente expresión es valida dentro de nuestro lenguaje: **va + 58 * r2 + p * 3**



El árbol se lee de abajo hacia arriba, cada elemento se va transformando en otra entidad no terminal por medio de la aplicación de diferentes reglas hasta llegar a ser una expresión. Como pudimos llegar a <EXPRESION> entonces sabemos que la sintaxis del texto del ejemplo, según el conjunto de reglas BNF dadas, efectivamente es una expresión bien escrita. Este árbol de parsing es ascendente, esto quiere decir que se parte del texto a verificar y se llega a la entidad Terminal que agrupa todo los conceptos (en este caso <EXPRESION>). A este terminal se lo llama ENTIDAD PRIVILEGIDADA o DISTINGUIDA.

El árbol de parsing descendente en cambio, comienza desde la entidad privilegiada y se desarrolla hasta obtener las entidades terminales que componen el texto a verificar. Si hiciéramos este árbol para el ejemplo anterior, comenzaríamos escribiendo en lo mas alto la entidad <EXPRESION> e iríamos aplicando las mismas reglas desde arriba hacia abajo hasta poder obtener la expresión que estamos verificando en el ejemplo.

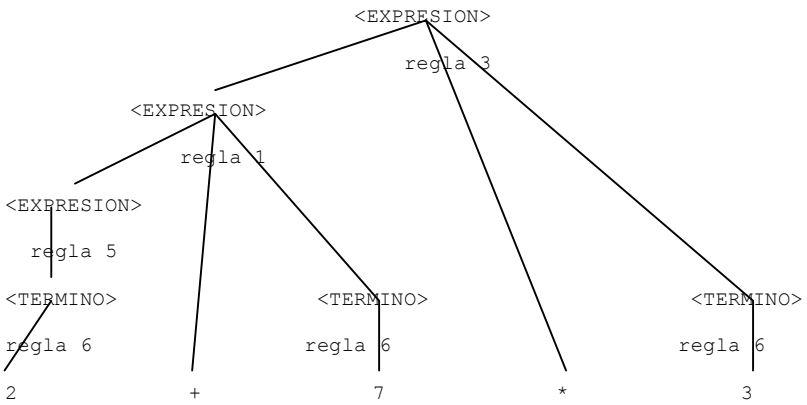
Si bien el principal objetivo de BNF es determinar si un programa esta bien o mal escrito según una sintaxis, también es utilizado a la hora de resolver el código del lenguaje en las instrucciones resultantes. Veamos un ejemplo de esto.

Tenemos la siguiente expresión: 2 + 7 * 3. Si hacemos primero la suma, el valor resultante seria 27, mientras que si realizamos primero la multiplicación el resultado es 23.

Supongamos la siguiente BNF:

1. <EXPRESION> → <EXPRESION> + <TERMINO>
2. <EXPRESION> → <EXPRESION> - <TERMINO>
3. <EXPRESION> → <EXPRESION> * <TERMINO>
4. <EXPRESION> → <EXPRESION> / <TERMINO>
5. <EXPRESION> → <TERMINO>
6. <TERMINO> → <VARIABLE> | <CONSTANTE>
7. ..... (para esta muestra simplificamos la definición de <VARIABLE> y <CONSTANTE> que son las mismas que ya mostramos pero deben estar siempre. Vamos a simplificarlo también en el árbol).

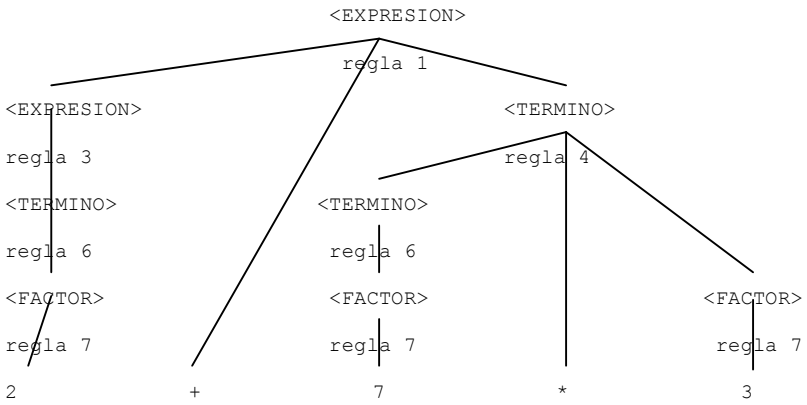
Hacemos el árbol de parsing para el ejemplo:



Como podemos apreciar, la formula es efectivamente una expresión. Observe el momento que aplicamos la regla 1, en esa regla estamos obteniendo el resultado de 2 + 7. Esto es, el compilador genera las instrucciones necesarias para realizar la suma antes que la multiplicación. Con la BNF utilizada, tanto el operador de suma, resta, multiplicación y división tienen la misma precedencia. Entonces, al tener la misma, se resuelve de izquierda a derecha.
Ahora, veamos otra BNF aplicada a la misma formula:

1. <EXPRESION> → <EXPRESION> + <TERMINO>
2. <EXPRESION> → <EXPRESION> - <TERMINO>
3. <EXPRESION> → <TERMINO>
4. <TERMINO> → < TERMINO> * <FACTOR>
5. < TERMINO> → < TERMINO> / <FACTOR>
6. < TERMINO> → <FACTOR>
7. <FACTOR> → <VARIABLE> | <CONSTANTE>
8. ...

Y hacemos el árbol de parsing:

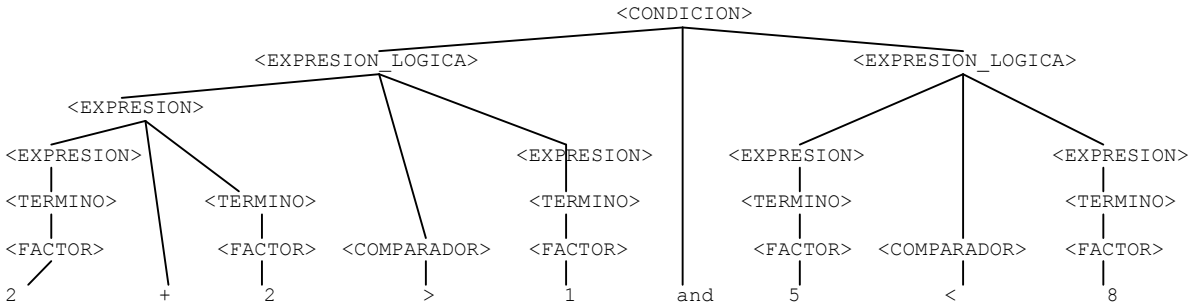


Al momento de aplicar la regla 4 el compilador genera las instrucciones necesarias para ejecutar la multiplicación. Esta se realiza antes que la suma a pesar de que se encuentra mas a la derecha. A esto se le llama que el operador de multiplicación tiene mas precedencia que el de la suma y viene dado por la BNF. Si observamos esta última, notamos que la multiplicación y la división están un nivel mas abajo que la suma y la resta.

Lo mismo ocurre con los operadores lógicos (AND, OR, etc) y los operadores de comparación (<, >, <=, >=, etc). Para contemplarlos, agregamos las siguientes reglas de BNF a la anterior que vimos:

<CONDICION> → <CONDICION> and <EXPRESION_LOGICA>
<CONDICION> → <CONDICION> or <EXPRESION_LOGICA>
<CONDICION> → <EXPRESION_LOGICA>
<EXPRESION_LOGICA> → <EXPRESION> <COMPARADOR> <EXPRESION>
<COMPARADOR> → = = | <= | < | >= | > | !=

Vemos como la comparación de expresiones está un nivel mas bajo que el uso de los operadores lógicos and y or. Hagamos el árbol de parsing de una condición:

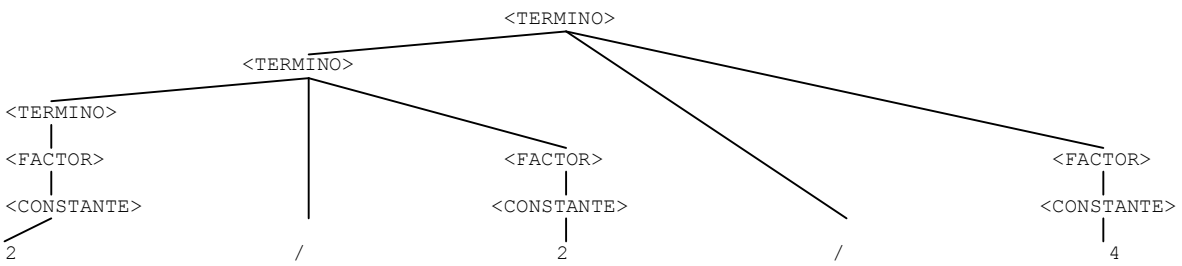


De esta forma, se ejecutan primero las expresiones (2 + 2), luego se ejecuta las expresiones lógicas (4 > 1 ; 5 < 8) y a ambos resultados se los procesa como condición (true and true = true).

Dijimos anteriormente que cuando los operadores aritméticos tenían la misma precedencia, las cuentas se resolvían de izquierda a derecha. También sabemos que resolver la cuenta 4 / 2 / 2 de izquierda a derecha retorna el resultado 1 mientras que hacerlo de derecha a izquierda es 4. De que depende que el compilador resuelva las cuentas en un determinado sentido?
Supongamos la siguiente BNF:

1. <TERMINO> → < TERMINO> * <FACTOR>
2. < TERMINO> → < TERMINO> / <FACTOR>
3. < TERMINO> → <FACTOR>
4. <FACTOR> → <VARIABLE> | <CONSTANTE>
5. ...

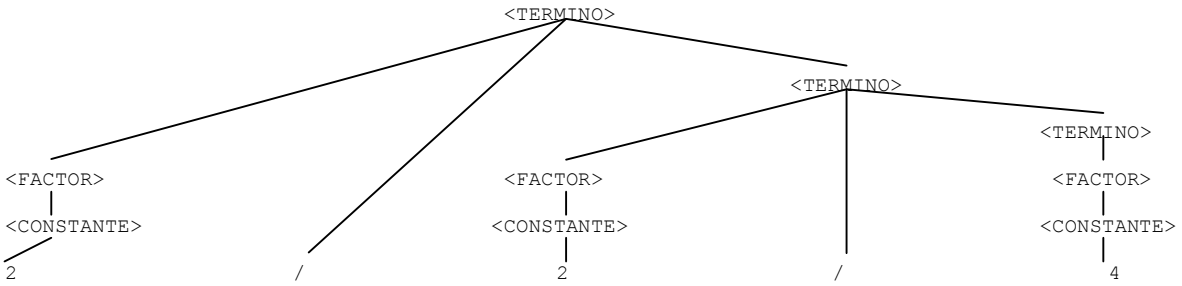
hacemos el árbol de parcing:



Como podemos ver, la primer división que se resuelve es 2 / 2. Esto en realidad es porque tuvimos que llevar el desarrollo del árbol por el lado izquierdo para que nos quede lo siguiente: <TERMINO> / <FACTOR> (en ese orden).
Ahora supongamos otra BNF:

5. <TERMINO> → < FACTOR> * <TERMINO>
6. < TERMINO> → < FACTOR> / <TERMINO>
7. < TERMINO> → <FACTOR>
8. <FACTOR> → <VARIABLE> | <CONSTANTE>
5. ...

hacemos el árbol de parcing:



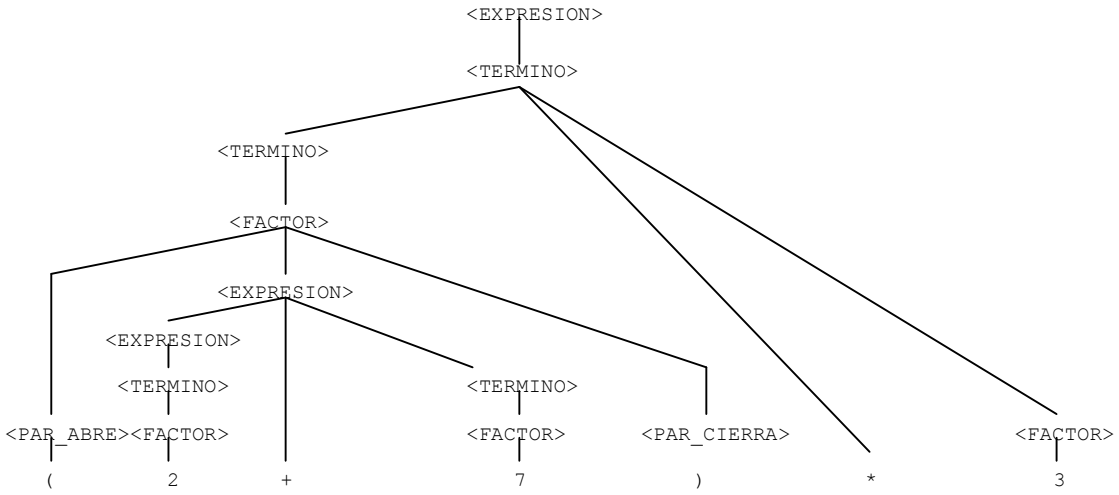
Ahora vemos lo contrario! Se resuelve primero la división 2 / 4. Esto es porque necesitamos llevar el árbol a una expresión como la siguiente: <FACTOR> / <TERMINO>. Si procediéramos a convertir el lado izquierdo en <TERMINO> nunca podríamos avanzar ya que no hay regla que contemple un <TERMINO> y un signo / a su derecha. Con estos ejemplos estamos viendo como la BNF influye en el sentido en el que se resuelven las operaciones aritméticas.

Esto último que vimos obviamente también se aplica a la suma y a la resta, y a cualquier par o grupo de operadores con la misma precedencia. De hecho, podemos hacer una BNF que resuelva las operaciones de multiplicación y división de izquierda a derecha y al mismo tiempo que resuelva las operaciones de suma y resta de derecha a izquierda en un nivel mayor.

Para terminar, vamos a incluir a los paréntesis en nuestra BNF. Veamos estas reglas:

1. <EXPRESION> → <EXPRESION> + <TERMINO>
2. <EXPRESION> → <EXPRESION> - <TERMINO>
3. <EXPRESION> → <TERMINO>
4. <TERMINO> → < TERMINO> * <FACTOR>
5. < TERMINO> → < TERMINO> / <FACTOR>
6. < TERMINO> → <FACTOR>
7. <FACTOR> → <VARIABLE> | <CONSTANTE>
8. <FACTOR> → <PARENTESIS_ABRE> <EXPRESION> <PARENTESIS_CIERRA>
9. <PAR_ABRE> → (
10. <PAR_CIERRA> →)

Como vemos, las reglas 8, 9 y 10 nunca las habíamos visto. Lo que estamos haciendo es agregándole una definición mas a la entidad <FACTOR>. Esta regla lo que hace es poner como contenido del paréntesis una expresión y todo lo que ello implica. Esto significa que una expresión compleja entre paréntesis puede ser un factor para otra operación fuera de los signos de paréntesis. Veamos un ejemplo rápido:



La resolución demuestra que para poder absorber los paréntesis con una regla se debe primero resolver lo que se encuentra entre estos dos símbolos para que sea una <EXPRESION>. Una vez que se tiene resuelta la parte interior, esto genera un <FACTOR> y todo vuelve a comenzar desde el nivel inferior de anidación. Gracias a esto, el compilador resuelve primero la formula matemática que se encuentra dentro de los paréntesis (que es una suma) y deja para el final la multiplicación (cuando normalmente, si no estuvieran los paréntesis, lo primero que resolvería es esta última operación).