Resumen de Lenguajes y Compiladores

Lucas Ponce de León

Universidad Nacional de La Matanza

ÍNDICE

ÍNDICE ***2***

PLANIFICACIÓN ***4***

LENGUAJES ***5***

Estructura de los Lenguajes de Programación *5*
Aspectos de un Lenguaje de Programación 5
Binding *5*
Tipo 5
Valor 6
Alcance 6
Almacenamiento / Tiempo de Vida 6
Resumen de Atributos de Binding de Variables 7
Tipos de Lenguajes *7*
Clasificación de los Lenguajes de Programación 7
Modelos de Ejecución 7
Uso de Memoria 7
Unidad o Bloque 9
Registros de Activación *9*
Cadena Dinámica 10
Estructura de Unidades 10
Cadena Estática 11
Tipos de Variables *12*
Variables Estáticas 12
Variables Semi-Estáticas 12
Variables Semi-Dinámicas 14
Variables Dinámicas 14
Variables Super-Dinámicas 15
Resumen de Tipos de Variables 15
Problemas con Punteros *15*
Conflicto de Tipos 16
Punteros Colgados (Danling Reference) 16
Basura (Garbage) 16
Gestión de Almacenamiento *17*
Garbage Collector en Lenguajes Dinámicos 18
Garbage Collector en Lenguajes Tipo Algol 18
Interacciones con el Sistema Operativo *19*
Sistemas Operativos con Particiones Fijas 19
Sistemas Operativos con Segmentos 19
Sistemas Operativos con Paginación y Memoria Virtual 19
Concurrencia y Hebras de Ejecución 20
Pasaje de Parámetros *21*
Clasificación según la Sintaxis 21
Clasificación según la Semántica 21
Evolución del Pasaje de Parámetros 22
Tipos de Datos y Conversiones *22*
Conversiones Implícitas en Algol (Coerciones) 23
Voiding 23
Widening 23
Rowing 24
Uniting 24

Desproceduring 24
Desreferencing 25
Control de Precisión *27*
Metalenguaje BNF *28*
Definición de una BNF 28
Árbol de Parsing 29
Gramática Ambigua 30
Asociatividad de la Gramática 30
COMPILADORES ***33***
Tabla de Símbolos *33*
Compilación Monolítica: 33
Compilaciones Separadas: 33
Vinculación Dinámica: 33
Bibliotecas Compartidas: 33
Tipos de Compiladores *33*
Cross Compiler 34
Autocompilador 34
Metacompiladores 34
Máquinas Objeto 34
Diagramas T 34
Estructura General de un Compilador *36*
Analizador Léxico *37*
Autómatas de Estados Finitos 37
Tratamiento de Errores 37
Analizador Sintáctico (Parser) *38*
Parsing Descendente (Top Down) *38*
Método “Recursivo Descendente” 39
Método “LL(1)” o “Predictivo Descendente” 40
Implementación del Método “LL(1)” o “Predictivo Descendente” 41
Parsing Ascendente (Buttom Up) *44*
Método “SLR” 44
Generación de Código Intermedio *51*
Árbol Sintáctico 52
Polaca Inversa 53
Tercetos 53
Cuartetos 53
Generación de Código Final *54*
Generación de Código Assembler 54
Traducción de Lista de Reglas a Árbol Sintáctico 54
Tipos de Datos en construcción del Árbol Sintáctico 57
Traducción de Árbol Sintáctico a Assembler 59
Seguimiento de Registros 61
Optimización de Código *63*
Reducción Simple 63
Redundancia 63
Reordenamiento de Instrucciones 63

Planificación

Día	Temas	Estado
1: Lunes 10	▪ Estructura de los Lenguajes de Programación ▪ Binding ▪ Tipos de Lenguajes ▪ Registros de Activación	▪ **RESUMIDO** ▪ **RESUMIDO** ▪ **RESUMIDO** ▪ **RESUMIDO**
2: Martes 11	▪ Tipos de Variables ▪ Problemas con Punteros ▪ Gestión de Almacenamiento ▪ Interacciones con el Sistema Operativo ▪ Pasaje de Parámetros	▪ **RESUMIDO** ▪ **RESUMIDO** ▪ **RESUMIDO** ▪ **RESUMIDO** ▪ **RESUMIDO**
3: Miércoles 12	▪ Tipos de Datos y Conversiones ▪ Control de Precisión ▪ Metalenguaje BNF	▪ **RESUMIDO** ▪ **RESUMIDO** ▪ **RESUMIDO**
4: Jueves 13	▪ Tabla de Símbolos ▪ Tipos de Compiladores ▪ Estructura General de un Compilador ▪ Analizador Léxico	▪ **RESUMIDO** ▪ **RESUMIDO** ▪ **RESUMIDO** ▪ **RESUMIDO**
5: Viernes 14	▪ Analizador Sintáctico ▪ Parsing Descendente (Top Down) ▪ Parsing Ascendente (Buttom Up)	▪ **RESUMIDO** ▪ **RESUMIDO** ▪ **RESUMIDO**
6: Sábado 15	▪ Generación de Código Intermedio ▪ Generación de Código Final ▪ Optimización de Código	▪ **RESUMIDO** ▪ **RESUMIDO** ▪ **RESUMIDO**
7: Domingo 16	▪ Repaso General ▪ Lectura del Resumen	
8: Lunes 17	▪ Repaso General ▪ EXÁMEN FINAL	

LENGUAJES

Estructura de los Lenguajes de Programación

Aspectos de un Lenguaje de Programación

- **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. Está 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:**Es un factor externo a la sintaxis y la semántica. La 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

El binding o ligadura es el momento exacto en el que conoce un atributo o propiedad de un elemento de cierto lenguaje.
Elementos del Lenguaje:

- 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 más 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, está 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 acá 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 más la memoria y se permite la recursividad.

Resumen de Atributos de Binding de Variables

Entonces, podemos decir que un binding es estático si está establecido antes del momento de la ejecución del programa y no puede ser cambiado más 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 de Binding de Variable	Estático	Dinámico
Valor	Constantes	Mayoría
Tipo	Mayoría	APL, LISP, J, SMALLTALK
Alcance	Mayoría	BASIC, APL, J
Almacenamiento	FORTRAN, COBOL	Mayoría

Tipos de Lenguajes

Clasificación de los Lenguajes de Programación

Si tiene almacenamiento estático →**LENGUAJE ESTÁTICO**
Si tiene almacenamiento dinámico
Si tiene tipo y alcance estáticos →**LENGUAJE TIPO ALGOL / ORIENTADO A LA PILA**
Si tiene tipo o alcance dinámico →**LENGUAJE DINÁMICO**

Modelos de Ejecución

- **Compilado:** El compilador 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.
- **Interpretado**: El intérprete toma un programa escrito en determinado lenguaje, "entiende el significado" y ejecuta cada instrucción directamente, sin cambiarlo.

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, FORTRAN.

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 cuánta memoria ocupan.

- **Lenguajes Dinámicos:**

Tienen un uso de memoria impredecible. 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 más 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 así, 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.
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 intérprete que esté ejecutando en memoria para ser ejecutados.

- **Lenguajes 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 sí 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 ACTIVACÍON. Hablaremos de este concepto más adelante.

Unidad o Bloque

Se define como un conjunto de instrucciones delimitadas de forma explícita 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 otraspalabras, la ejecución llega al lugar donde se encuentra declarado. Debido a que no puede ser llamado, no esrecursivo.
- Con Nombre: Tienen nombre, son funciones, procedimientos, etc. Son invocados por una instrucción ypueden ser recursivos.

Registros 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.

Programa ejecutable
Reg. Act. A
Reg. Act. B
Reg. Act. C

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
```


Los offsets son asignados al momento de la compilación y quedan fijos en el código del programa. De esta formapodemos ver claramente que el nombre de la variable no queda disponible en el momento de ejecución en loslenguajes de tipo pila. Esto solo ocurre en los lenguajes Dinámicos.

Cadena Dinámica

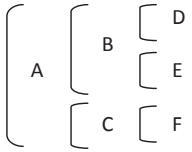
Antes de explicar este concepto, vamos a explicar la problemática. En cada momento, el programa mantiene al registro BP apuntando a la base del registro de activación activo. Cuando se produce un llamado a otro bloque, antes de realizar el salto al código correspondiente, se debe apuntar al nuevo registro 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.
```

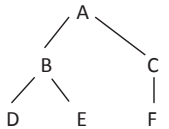
Mas adelante, con la ejecución del programa llega el momento de retornar de una función. Lo que se debe realizar en ese momento es volver atrás el BP, es decir, apuntar al registro de activación anterior. Pero el problema es que en este caso, una función no puede saber por quién ha sido llamada, con 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.
Entonces la CADENA DINÁMICA es una sucesión de punteros que va desde el registro de activación activo hasta el primero, pasando por todos dentro de la pila de llamadas.

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 varía luego de la compilación. Las unidades pueden estar anidadas hacia adentro o independientes.



Estructura de Unidades



Árbol de Anidamiento

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 D y E. 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.

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 qué distancia esta cada variable (dentro del árbol de anidamiento) según la unidad donde se encuentra la operación.
El concepto de arco es la cantidad de saltos atrás que debe hacer para encontrar la variable dentro del árbol de anidación.
Supongamos el ejemplo:

Variable	Arcos
Z	3
X	2
Y	1

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 pseudocódigo en Assembler que de cómo se compilaría la suma y más 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
```

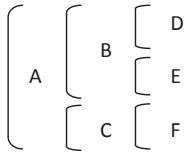
Cadena Estática

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 ESTÁTICA. 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.

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.

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.



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 más ya que estamos en el nodo raíz.
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

=

	A	B	C	D	E	F
A	1	0	0	-	-	-
B	2	1	1	0	0	-
C	2	1	1	-	-	0
D	3	2	2	1	1	-
E	3	2	2	1	1	-
F	3	2	2	-	-	1

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

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).

Construcción de la Cadena Estática:
Ahora que sabemos que unidad puede llamar a cual otra, podemos aclarar cómo 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 pre construida.

Algoritmo para enlazar cadenas de un nuevo Registro de Activación:

1) Enlazar Cadena Dinámica (CD)
2) Enlazar Cadena Estática (CE)
    a. Realizar un respaldo de BP (que tiene la base actual)
    b. Bajar un arco por la CD
    c. Bajar un arco por la CE por cada nivel de anidamiento que se tenga
    d. Copiar BP a donde apunta la CE
    e. Restaurar BP a la base

Tipos de Variables

Variables 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 FORTRAN.

Variables Semi-Estáticas

Tienen lugar variable en distintas ejecuciones y tamaño fijo. Son propias de los Lenguajes Tipo Algol. El registro de activación que las contiene varía de lugar, pero lo que no varía es el offset de la variable dentro del mismo.

Ejemplo 1:
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 dónde empieza y donde termina porque tiene los límites especificados al momento de la definición (almacena todos los elementos del array de forma contigua).
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 fórmula: `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 así 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.

Ejemplo 2:
Otra variable semi-estática 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:

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

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

Variables Semi-Dinámicas

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 semi-estática. 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 semi-estáticas por tener un offset fijo relativo al comienzo del R.A.
Para compilar un acceso al arreglo, se maneja como con los arreglos semi-estáticos (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.

Variables 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 está 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 semi-estática, 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 explícita 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 undescriptor 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 enforma 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 semi-estáticas y dinámicas anónimas. En ADA solo existen variablessemi-estáticas, dinámicas anónimas y variables semi-dinámicas.

Variables Súper-Dinámicas

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 tipos 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.

Resumen de Tipos de Variables

Variable Estática	Variable Semi-Estática	Variable Semi-Dinámica	Variable Dinámica	Variable Super-Dinámica
Ubicación Fija	Ubicación Variable	Ubicación Variable	Ubicación Variable	Ubicación Variable
Tamaño Fijo	Tamaño Fijo	Tamaño Variable (se conoce previamente)	Tamaño Variable	Tamaño Variable
Lenguajes Estáticos	Lenguajes Tipo Algol	Lenguajes Tipo Algol	Lenguajes Tipo Algol	Lenguajes Dinámicos
Datos en Stack	Datos en Stack	Descriptor en Stack Datos en Stack	Descriptor en Stack Datos en Heap	Descriptor en TDS Datos en Heap
	Ej.: Punteros Arrays de Tamaño Fijo	Ej.: Arrays de Tamaño Variable		Tipos Dinámicos

Problemas con Punteros

Como vimos, dentro del registro de activación puede haber punteros hacia otras estructuras como pueden ser a variables semi-estáticas, semi-dinámicas, 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.

El motivo por el cual se producen problemas es porque el programador puede hacer una mala administración de los mismos. Los punteros que administra el programador son:

- Punteros que apuntan a otra variable semi-estática.
- Variables dinámicas anónimas que tienen un puntero referenciando a un bloque de datos en el HEAP.
- Punteros que apuntan a variables semi-dinámicas.

Los problemas que pueden surgir son:

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 da 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.
Más 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 (Danling Reference)

En este caso el programador manipula los punteros de tal forma que luego de tomar un bloque de datos, lolibera 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 esté 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.

Basura (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:

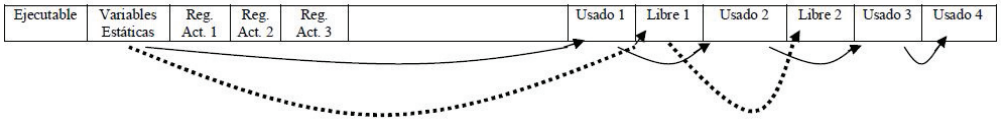
```
p = malloc(100);
q = malloc(200);
p = q;
```

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.

Gestión de Almacenamiento

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 más rápida que la pila.

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

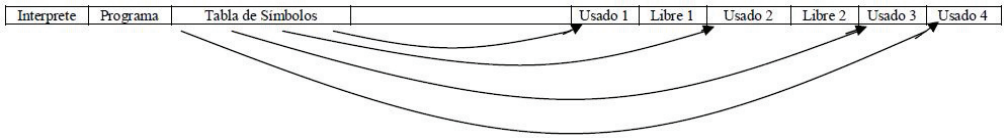


Aquí se representa la partición fija de la memoria donde está contenido el programa. Sobre la izquierda se encuentra el ejecutable, las variables estáticas y la pila de registros de activación, a la derecha se ubica el Heap. Existen dos estructuras que son utilizadas para mantener los espacios usados 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 más con cambios de tipos (por ejemplo, que una variable pase a ser de un nuevo tipo más 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 en los lenguajes dinámicos 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:

1. El intérprete revisa la tabla de símbolos mirando los punteros y elige el que apunte a la dirección más alta de memoria.
2. Se pregunta si está al fondo de la partición fija.
   a. Si lo está, simplemente la marca como "ya revisada".
   b. Si no lo está, corre al fondo el bloque de datos, obviamente sin pisar otros bloques que se encontraran más atrás. La marca como "ya revisada".
3. 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:

1. Recorre la lista de usados y a cada uno le coloca una marca.
2. 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.
3. Recorre la cadena de usados liberando los bloques que conservan la marca.
4. OPCIONAL: Se puede aplicar un algoritmo de compactación con el de Lenguajes Dinámicos.

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 diseñó un algoritmo (Inversión de Punteros) 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. 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.

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

Interacciones con el Sistema Operativo

Sistemas Operativos con Particiones Fijas

En estos Sistemas el programa corre en una partición fija de memoria que no varía 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

Sistemas Operativos 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 cuánta 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, así 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 pérdida 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.

Sistemas Operativos 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 más espacio difícilmente la pila y el Heap se choquen. La fragmentación interna podría verse como mucho más 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.

Administración de Memoria	L. Estático	L. Tipo Algol	L. Dinámico
Particiones Fijas	OK	Puede fallar	Puede fallar
Segmentos	OK	Puede fallar	Puede fallar
Paginación y Memoria Virtual	OK	OK	OK

Concurrencia y Hebras de Ejecución

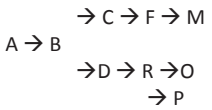
Sabemos que un Sistema Operativo puede manejar concurrencia, un ejemplo de esto es UNIX. También sabemos que existen lenguajes que ejecutan tareas concurrentes, por ejemplo ADA.

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:

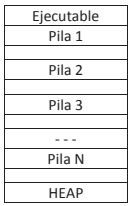
<table>
<tr><th></th><th>Lenguaje Sin Concurrencia</th><th>Lenguaje Concurrente</th></tr>
<tr><td>S.O. Sin Concurrencia</td><td>Obviamente, no existe la concurrencia en este sistema.</td><td>Existe la concurrencia administrada por el lenguaje. Este es el que implementa los métodos de sincronización y lo que se conmuta es entre procedimientos del mismo programa.</td></tr>
<tr><td rowspan="4">S.O. Concurrente</td><td rowspan="4">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.</td><td>El S.O. y el Lenguaje no se conocen</td></tr>
<tr><td>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.</td></tr>
<tr><td>El S.O. y el Lenguaje se conocen (HEBRAS)</td></tr>
<tr><td>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.</td></tr>
</table>

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:



Con esta situación, ya no se puede utilizar una pila como la estábamos utilizando antes porque pierde su carácter de LIFO.

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 primera 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

Tipo de Pasaje		Descripción	Ejemplo
Asociación Posicional	**Sin faltantes**	Tiene que ser uno a uno, no hay faltantes. En cada llamado existe la misma cantidad deparámetros reales que de formales.	
	Con faltantes al final	El lenguaje permite que la llamada a la función no tenga la misma cantidadde parámetros en la definición formal pero los que falten son los que se encuentran alfinal.	funcionA (a, b, c) **funcionA (int x, int y, int z, int w)**
	Con faltantes en cualquier lugar	El lenguaje ofrece una forma de indicar que parámetros son losque se le pasa el valor y cuáles no.	funcionA (_ , _ , 3) **funcionA (int x, int y, int z)**
Asociación Explícita		En la llamada se indica explícitamente que valor real se asigna a que parámetro formal.	funcionA (x as 3, y as 4) **funcionA (int x, int y)**
Asociación Anónima		La unidad recibe una cantidad variable de parámetros a partir de algún mecanismoimplementado por el lenguaje.	funcionX(55, a) funcionX(2, 3, 8, 4) **funcionX (float b, ...)**
Valores por defecto		Los parámetros pueden tener un valor por defecto si es que en la llamada no se le pasa valor.	funcion(int x, int y, float z = 0.5);

Clasificación según la Semántica

Tipo de Pasaje	Descripción
Por Referencia	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.
Por Nombre	Se genera un nuevo código en memoria denominado "thunk" en el cual el compilador reemplaza los parámetros formales por los reales textualmente.Esto lo usa Algol y hoy día en las Macro.
Por 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.
Por Copia Resultado	Realiza exactamente lo contrario al anterior, asignando los valores resultantes sobre los parámetros reales al finalizar la ejecución de la unidad llamada.
Por Copia Valor/Resultado	Realiza ambas cosas.

Evolución del Pasaje de Parámetros

Al principio cuando se creóel compilador de FORTRAN 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.

Más tarde se creóALGOL 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 así:

```
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.

Tipos de Datos y Conversiones

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 acá en más, 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 ycada uno de sus bits significa 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 así? 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 tipos 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".

Conversiones Implícitas en Algol (Coerciones)

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.
Ejemplo:

```
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.
Ejemplo:

```
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.
Ejemplo:

```
[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 más 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 más grande de los valores y un byte más 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 lo 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 así 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 lo siguiente:

```
tmp := xx; // El proceso 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 así 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

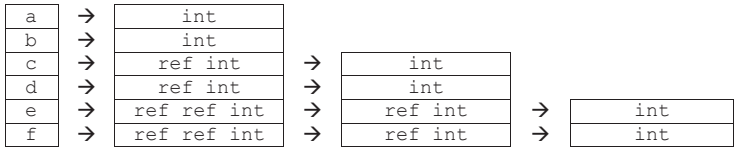
Para ver este concepto primero debemos explicar lo siguiente. Vamos a declarar las siguientes variables en Algol:

```
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. Acá vemos el mismo código pero en lenguaje C para que se entienda mejor:

```
int a, b, *c, *d, **e, **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.

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.	**1 desreferencing**
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.	**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.	**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.	**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.	**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.	**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.	**IMPOSIBLE**
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.	**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.	**1 desreferencing**

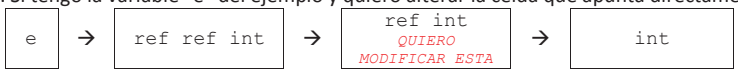
La idea de Algol era matar la diferencia entre la dirección de un valor y el contenido del mismo. Esto dio confusión a los programadores. A partir de Algol los lenguajes modernos tienen un desreferencing implícito.

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 así 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.

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 desreferencing. 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 puede depender de la forma que tiene el mismo de representar los valores de coma flotante.
Por ejemplo:

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

Para saber qué 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, alser una representación binaria existen ciertos números que son periódicos pero en decimal no lo son. Porejemplo, en decimal el 0.1 es exacto y en binario ese mismo número es periódico. Por eso, en el código deejemplo, cuando se realiza la operación "x * 10.0", al tener "0.1" esto no da exacto sino un número periódicoque 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 decimalcon cuatro bits. Por ejemplo, el valor 13.6 en BCD está 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 unafiel representación del mismo. La contra de esto es que los valores ocupan más espacio de almacenamiento(simple de 32 bits y doble de 80 bits) y que las operaciones matemáticas son mucho más lentas.Para profundizar sobre IEEE decimos por ejemplo que "5 * 1/5" no nos da 1 como pensamos, porque el 1/5 esperiódico en binario.

Lenguaje	Formato
Cobol	BCD (Porque necesita exactitud)
Fortran	Pre IEEE
C	Ambas (pero la misma en todo el programa)
Pascal	IEEE
Ada	Ambas (pero dirigido a la sintaxis)

Ejemplo en ADA:

```
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 valor12.30 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.

Metalenguaje BNF

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

Definición de una BNF

Elementos utilizados para definir un lenguaje en BNF:

- Flecha de Definición (→): La fecha a la derecha significa "está definido como". Se usa de la siguiente manera:
  A →B donde A es el objeto que está 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 ítem 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.
- Elemento Distinguido (Start Symbol): Es el símbolo que indica el comienzo del lenguaje. Además es el No Terminal más abarcativo de todos.

Además cabe resaltar que como son **Reglas Declarativas**, no importa el orden en que vayan aplicadas.
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 (o componente léxico) 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.

Árbol de Parsing

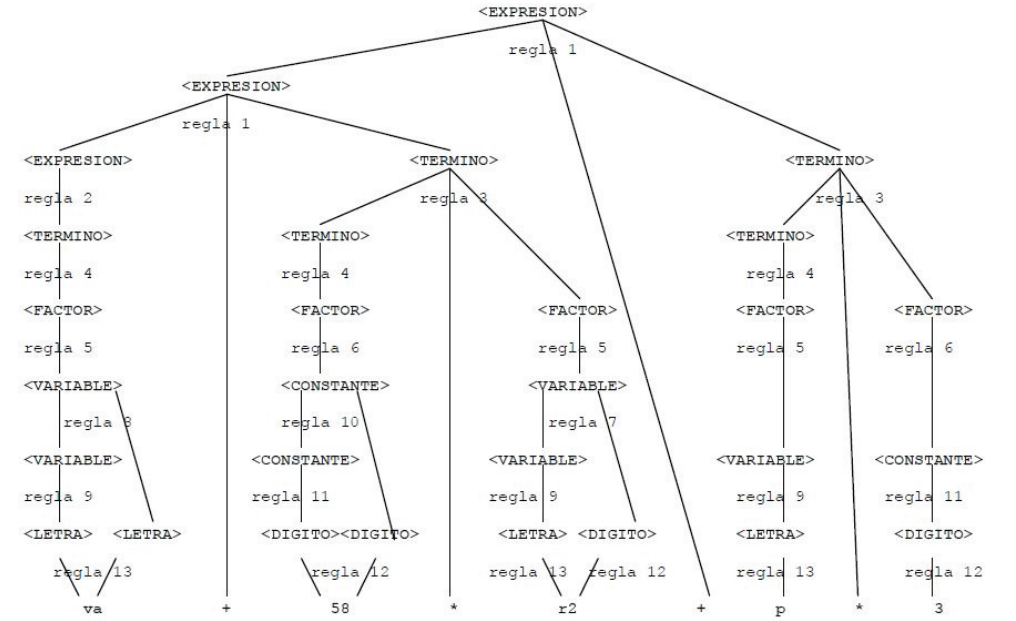
Es una estructura que se realiza para comprobar si una expresión está correctamente escrita según las reglas BNF dadas, es decir, se utiliza para que dado un programa escrito en determinado lenguaje y las reglas BNF que componen su sintaxis, poder saber si el programa es válido o no.

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 válida dentro de nuestro lenguaje: va + 58 * r 2 + 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 más 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.

Gramática Ambigua

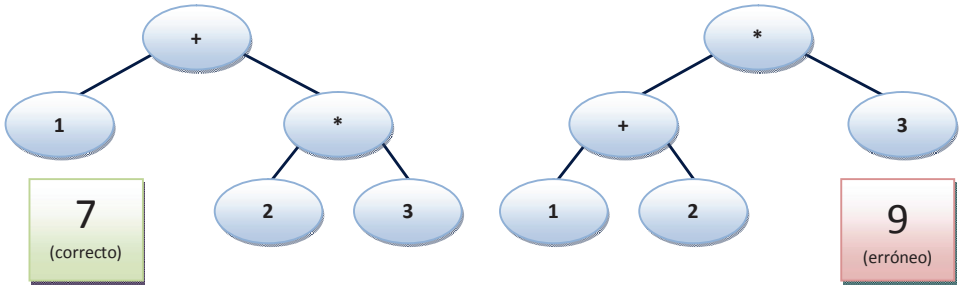
Una gramática es ambigua para cuando para un mismo código, se pueden generar dos ó más árboles de parsing distintos que den resultados diferentes.

Si tenemos la siguiente expresión: 1+2*3. Si hacemos primero la suma, el valor resultante sería 9, mientras que si realizamos primero la multiplicación el resultado es 7 (el que se espera).

Supongamos la siguiente BNF:

1. <EXPRESION>→<EXPRESION> + <EXPRESION>
2. <EXPRESION>→<EXPRESION>-<EXPRESION>
3. <EXPRESION>→<EXPRESION> * <EXPRESION>
4. <EXPRESION>→<EXPRESION> / <EXPRESION>
5. <EXPRESION>→<VARIABLE>
6. <EXPRESION>→<CONSTANTE>

Entonces para una misma sentencia, existen dos árboles de parsing posibles:



Por lo cual, decimos que la gramática es ambigua.

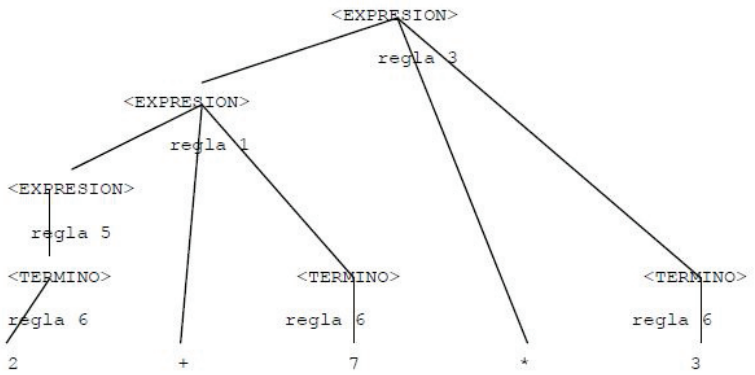
Asociatividad de la Gramática

Si bien el principal objetivo de BNF es determinar si un programa está 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 sería 27, mientras que si realizamos primero la multiplicación el resultado es 23.

Supongamos la siguiente BNF:

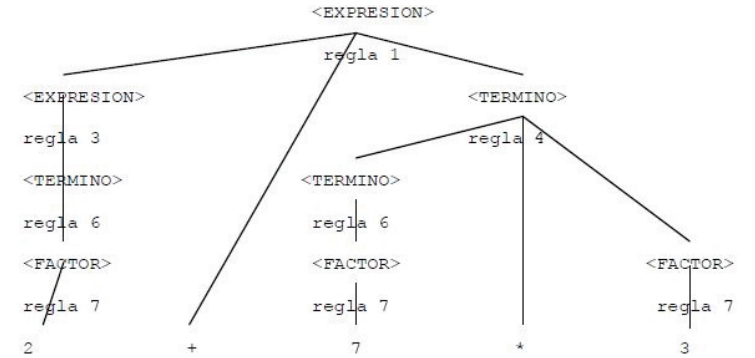
1. <EXPRESION>→<EXPRESION> + <TERMINO>
2. <EXPRESION>→<EXPRESION><TERMINO>
3. <EXPRESION>→<EXPRESION> * <TERMINO>
4. <EXPRESION>→<EXPRESION> / <TERMINO>
5. <EXPRESION>→<TERMINO>
6. <TERMINO>→<VARIABLE> | <CONSTANTE>



Como podemos apreciar, la fórmula 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 fórmula:

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>



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 más a la derecha. A esto se le llama que el operador de multiplicación tiene más 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 más abajo que la suma y la resta.

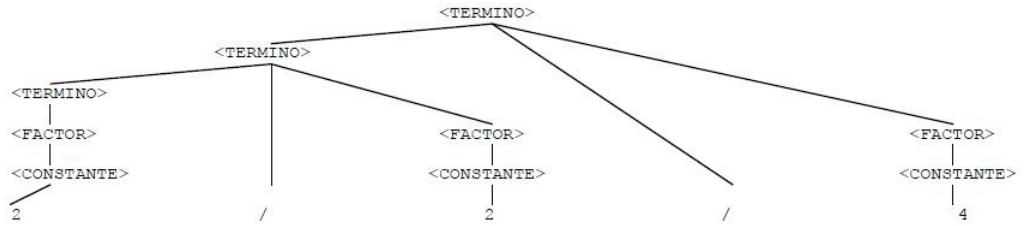
Lo mismo ocurre con los operadores lógicos (AND, OR, etc) y los operadores de comparación (<, >, <=, >=, etc).

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 qué 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>

El árbol crece hacia la izquierda

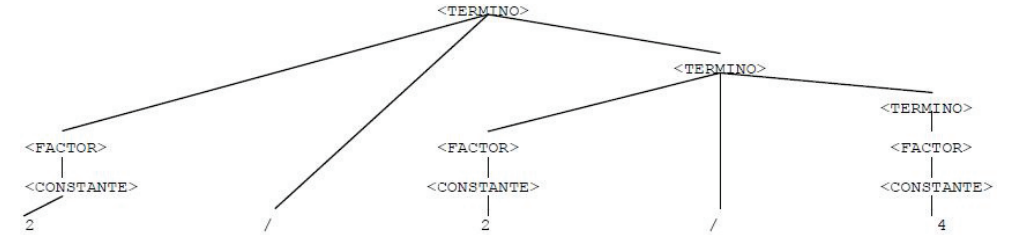


Como podemos ver, la primer división que se resuelve es 2 / 2. Esto en realidad es porque tuvimos que llevar el desarrollo del árbol por el lado izquierdo para que nos quede lo siguiente: <TERMINO> / <FACTOR> (en ese orden).

Ahora supongamos otra BNF:

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

El árbol crece hacia la derecha



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.

COMPILADORES

Tabla de Símbolos

La tabla de símbolos es una estructura que es utilizada tanto en los lenguajes dinámicos (en tiempo de ejecución) como en los tipo Algol (solo en tiempo de compilación, sirve de apoyo para el proceso pero una vez que comienza la ejecución la tabla desaparece).
Así se comporta la tabla de símbolos en los distintos tipos de compilaciones:

Compilación Monolítica:

Se le llama así cuando la compilación del programa tiene todos los datos suficientes para llevarse a cabo por completo. Por ejemplo, un programa que dispone de todos los módulos que se utilizarán en la ejecución. En este caso la tabla de símbolos se usa en compilación y luego se descarta.

Compilaciones Separadas:

Se da cuando los lenguajes permiten generar archivos de código separados.Para esta forma de compilar no se puede descartar por completo la tabla de símbolos luego de la compilación ya que una referencia desde uno de los fuentes puede estar apuntando a otro.
Por ejemplo, si tengo un procedimiento "calcularPrecio" en el fuente 2 y es llamado desde el fuente 1, al momento decompilar este último debo saber dónde se encuentra el procedimiento. La información sobre el procedimiento se guarda en el mismo binario, es decir, el binario 2 mantiene la referencia "calcularPrecio" del procedimiento ya compilado para que una vez que se ejecute el Linker este pueda resolver estas referencias pendientes.

Los elementos que interesan mantener como referencias no resueltas para un binario determinado son:

- Las estructuras que tiene el binario y que se usan externamente.
- Las estructuras externas que son usadas por este binario.

De esta forma se dice que la tabla de símbolos sobrevive parcialmente en las compilaciones de este tipo(queda embebida en los binarios generados en la compilación pero una vez ejecutado el Linker no queda nada de ella).

Vinculación Dinámica:

Es el caso de programas con uso de DLL (en Windows) o ELF (en Linux). En este modelo existen dos linkeos, el primero en el momento de generar el ejecutable. Luego de este proceso en el ejecutable quedan referencias no resueltas a procedimientos de las bibliotecas del Sistema Operativo.
Luego al momento de ejecutar el programa el Sistema Operativo realiza un segundo linkeo donde resuelve dichas referencias y de esta forma se ejecuta el programa. De esta forma existe parte de la tabla de símbolos en el ejecutable en disco que hacen referencias a estructuras de la biblioteca y que son resueltas por el linker del Sistema Operativo.

Bibliotecas Compartidas:

En este modelo ocurre algo parecido que en el anterior solo que el linker 2 no resuelve las referencias sino hasta que el procedimiento es llamado por el programa. El flujo de ejecución de un programa puede llevar a no resolver nunca una referencia para un procedimiento determinado.

Tipos de Compiladores

Vamos a hablar de algunos conceptos generales sobre los Compiladores. Lo primero que hay que decir es que un compilador no necesariamente tiene como salida el código ejecutable. Por ejemplo, el compilador de ADA genera un fuente en C que luego es compilado por otro aplicativo.

Cross Compiler

Es un compilador que corre en una máquina y la salida está preparada para otra máquina. Por ejemplo un compilador de C en SUN para una máquina IBM.

Autocompilador

Es un compilador de un lenguaje escrito en su mismo lenguaje. Por ejemplo un compilador de Pascal escrito en Pascal.

Metacompiladores

Son compiladores que tenían como objetivo crear compiladores. Es una especie de solución automatizada para generar compiladores automáticos. Al metacompilador debía proporcionársele:

- Sintaxis del nuevo lenguaje.
- Semántica del nuevo lenguaje.
- Información del Sistema Operativo (Ej.: como llamar a los servicios)

Este experimento no se logró hacer con éxito pero si se pudo generar Analizadores Léxicos y Sintácticos automáticamente. El producto más conocido que realiza esta tarea es YACC.

Máquinas Objeto

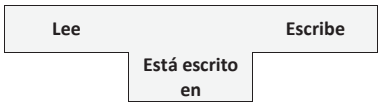
En una época que faltaba recurso humano que trabaje creando compiladores definieron una máquina imaginaria y luego crearon un compilador de un código que "corría" en esa máquina inexistente. Luego crearon intérpretes de esa máquina que ejecutaban sobre máquinas reales. Esta idea finalmente murió pero tiempo después resurgió con la aparición de Java.

Diagramas T

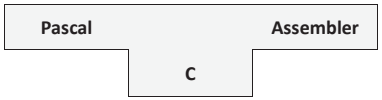
Para explicar en qué consiste un diagrama T primero vamos a hablar de ciertos conceptos. En la construcción de un compilador hay tres lenguajes:

1) El lenguaje que el compilador entiende y que es escrito por el programador (el fuente origen).
2) El código en el que el compilador escribe (el resultado de la compilación).
3) El lenguaje en el que está escrito el compilador.

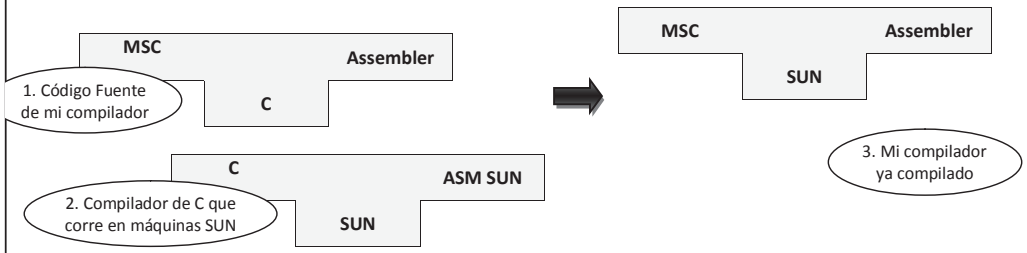
Este escenario se representa con un diagrama T:



Por ejemplo, un compilador de lenguaje Pascal que está escrito en C y genere una salida en Assembler se representa con el siguiente diagrama T:

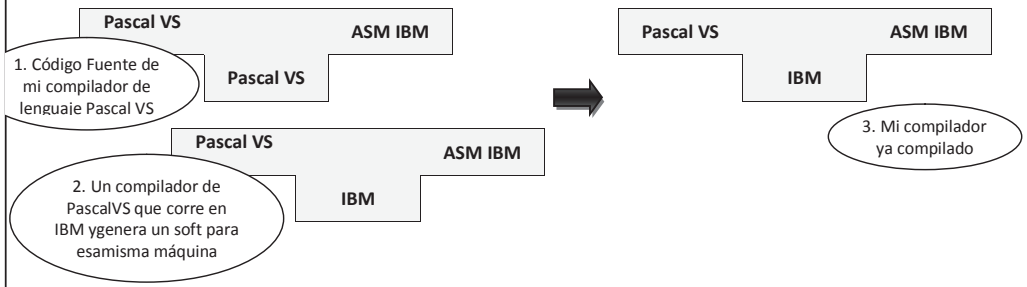


Podemos hacer un diagrama T del compilador que estamos creando y también podemos hacer el diagrama T del compilador con el que estamos compilando nuestro compilador.
Ejemplos (MSC es el nombre de un lenguaje determinado).



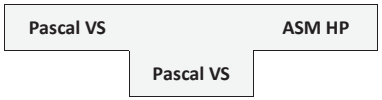
En este esquema se representa la acción de agarrar el código fuente del compilador que estoy creando (primera T) y compilarlo con un compilador tradicional de C que corre en máquinas SUN (segunda T) y obtener como resultado mi compilador pero en binario listo para usar (tercer T). Observe que cuando hablamos de compiladores ya compilados se le indica en que máquina corre en vez de poner en que lenguaje fue escrito porque una vez compilado el programa poco importa en qué lenguaje fue creado.

A continuación mostramos un ejemplo con un autocompilador ya que fue escrito en Pascal VS y lee fuentes de Pascal VS:



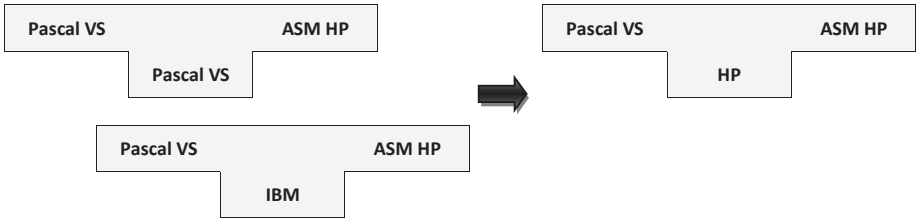
Supongamos que queremos que nuestro compilador corra en máquinas HP y genere código HP.

1. Reescribimos la salida de nuestro compilador y para que genere código HP, quedando:



2. Compilamos nuestro desarrollo con nuestro compilador de Pascal VS para IBM dándonos como resultado:

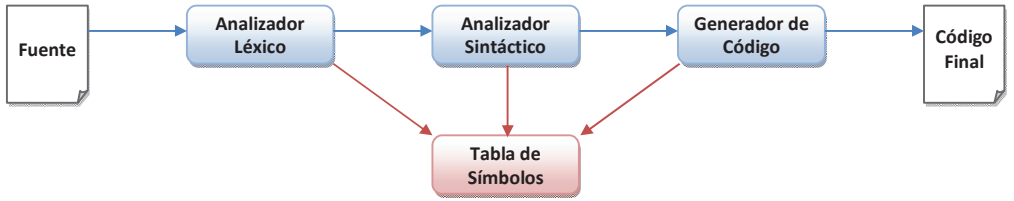
3. Compilamos el fuente de nuestro compilador con nuestro compilador en binario que obtuvimos en el paso anterior, nos da como resultado nuestro compilador de Pascal VS corriendo en HP.



Con lo cual, para hacer esto se tuvo que compilar dos veces el fuente de nuestro autocompilador, la primera con un compilador común y corriente de nuestro lenguaje y la segunda con nuestro propio autocompilador resultante del paso anterior. Esa es la ventaja de escribir un autocompilador, simplifica la tarea de portarlo a otra máquina.

Estructura General de un Compilador

El proceso de compilación se puede dividir en las siguientes etapas:



El archivo fuente, no es ni más ni menos que un conjunto de caracteres.

El analizador léxico divide el programa en fragmentos reconocibles a partir de expresiones regulares o de un autómata de estados finitos. Por ejemplo:

```
A := B + 10 * A;
```

Este sería el código fuente que debe reconocer el analizador léxico y a continuación se muestra que es lo que sale de dicho proceso:

```
<id A><símbolo de asignación><id B><operador +><constante 10><operador *><id A><fin de línea>
```

Es decir, ignora espacios, comentarios y determina cada uno de los elementos que existen en el fuente arrojando error si no reconoce alguno. Cada uno de estos elementos reconocidos se denomina Token o Componente Léxico. Se utiliza una tabla de Tokens donde se enumeran cada uno de ellos.
En definitiva, la salida del subproceso del Analizador Léxico es una tira de Tokens reconocidos por medio de sus códigos en la tabla de Tokens.

El Analizador Sintáctico construye el árbol de parsing utilizando las reglas BNF del lenguaje. La salida del Analizador Sintáctico es la lista de reglas que cumplieron al construir dicho árbol.

El generador de código puede dividirse en tres etapas:

- Generador de Código Intermedio: Es opcional. Convierte la lista de reglas que le otorga el analizador sintáctico a una notación intermedia (que puede ser Árbol Sintáctico, Tercetos o Polaca Inversa). La ventaja de generar código intermedio es que incrementa la portabilidad ya que con este código luego se

puede traducir fácilmente a diferentes códigos finales (por ejemplo, distintas versiones de Assembler para IBM, HP, etc.)
- Generador de Código Final: Es el producto final del compilador, es la única etapa obligatoria del generador de código.
- Optimizador de Código: Esta etapa también es opcional. Tiene como objetivo obtener un código final más eficiente.

El código final, puede estar escrito por ejemplo en Assembler, C ó Bytecode. No necesariamente debe ser un código ejecutable.

Analizador Léxico

Vimos que la primera fase del análisis es el análisis léxico. El principal objetivo del analizador léxico es leer el flujo de caracteres de entrada y transformarlo en una secuencia de componentes léxicos (o tokens) que utilizará el analizador sintáctico.
Al tiempo que realiza esta función, el analizador léxico se ocupa de ciertas labores de "limpieza". Entre ellas está eliminar los blancos o los comentarios. También se ocupa de los problemas que pueden surgir por los distintos juegos de caracteres o si el lenguaje no distingue mayúsculas y minúsculas.
Para reducir la complejidad, los posibles símbolos se agrupan en lo que llamaremos categorías léxicas. Tendremos que especificar qué elementos componen estas categorías, para lo que emplearemos expresiones regulares. También será necesario determinar si una cadena pertenece o no a una categoría, lo que se puede hacer eficientemente mediante autómatas de estados finitos.
Entonces podemos decir que las funciones principales del Analizador Léxico son:
1. Transformar la cadena de caracteres de entrada en componentes léxicas.
2. Llevar un control en una tabla de símbolos para reconocer las componentes léxicos no únicas como los identificadores y constantes.
3. Quitar todo lo inservible del código, como espacios, tabulaciones, etc.
4. Realizar el tratamiento de errores apropiado.

Autómatas de Estados Finitos

Estos son máquinas formales que consisten en un conjunto de estados y una serie de transiciones entre ellos. Para analizar una frase, el autómata se sitúa en un estado especial, el estado inicial.
Después va cambiando su estado a medida que consume símbolos de la entrada hasta que esta se agota o no se puede cambiar de estado con el símbolos consumido. Si el estado que alcanza es un estado final, decimos que la cadena es aceptada; en caso contrario, es rechazada.
Distinguimos dos tipos de autómata según cómo sean sus transiciones. Si desde cualquier estado hay como mucho una transición por símbolos, el autómata es determinista. En caso de que haya algún estado tal que con un símbolo pueda transitar a más de un estado, el autómata es no determinista. Nosotros trabajaremos únicamente con autómatas deterministas.

Para implementar el autómata de estados finitos, se suelen utilizar dos matrices:
- Matriz de transición de estados: define a qué estado cambiar en función del estado actual y del próximo carácter a procesar.
- Matriz de punteros a funciones: define qué procedimiento ejecutar en función del estado actual y del próximo carácter a procesar.

Tratamiento de Errores

Si el autómata finito no puede transitar desde un determinado estado y no ha pasadopor estado final alguno, se ha detectado un error léxico. Tenemos que tratarlo de alguna manera adecuada.

Desafortunadamente, es difícil saber cuál ha sido el error. Podemos preguntarnos la causa del error ante la entrada "a.3". Podrá suceder que estuviésemos escribiendo un real y hubiésemos sustituido el primer dígito por la a. Por otro lado, podrá ser un rango mal escrito por haber borrado el segundo punto. También podrá suceder queel punto sobrase y quisiéramos escribir el identificador "a3". Como puedes ver, aún en un caso tansencillo es imposible saber qué ha pasado.
Una posible estrategia ante los errores será definir una "distancia" entre programas y buscar elprograma más próximo a la entrada encontrada. Desgraciadamente, esto resulta bastante costosoy las ganancias en la práctica suelen ser mínimas.
Otra estrategia más simple es la siguiente:
1. Emitir un mensaje de error y detener la generación de código.
2. Devolver al flujo de entrada todos los caracteres leídos desde que se detectó el último componenteléxico.
3. Eliminar el primer carácter.
4. Continuar el análisis.

Siguiendo estos pasos, ante la entrada "a.3", primero se emitirá un identificador, después se señalael error al leer el "3". Se devolverán el tres y el punto a la entrada. Se eliminará el punto y se seguiráel análisis emitiendo un entero.

Analizador Sintáctico (Parser)

Es el elemento del compilador que se encarga de verificar la semántica del lenguaje.
Luego de que el Analizador Léxico haya procesado el código de entrada, se ejecuta el Analizador Sintáctico. Este toma la tira de tokens generada y aplicando la gramática devuelve una lista de reglas que se aplicó para crear el árbol de parsing. Este árbol es abstracto, es decir, no se crea en memoria sino que se expresa a través de la lista de reglas aplicadas que son output al proceso del Analizador Sintáctico.

Además, para un cierto programa dado, pueden existir varios conjuntos de reglas que son válidas. A pesar de esto, es deseable que no haya más de un árbol de parsing para el mismo programa, ya que si lo hay, se dice que la gramática es ambigua.

En resumen, el analizador sintáctico se encarga de:
- Controlar que la "tira de tokens" que se recibe estén escritas correctamente y tengan sentido semántico, de acuerdo a las reglas sintácticas que están en la BNF.
- Revisar, cambiar, agregar o quitar elementos de la Tabla de Símbolos.
- Generar la lista de reglas, el cual es el objetivo principal.

Básicamente existen dos metodologías diferentes para generar el árbol de parsing:
- **Parsing Descendente (Top Down):** En el que se parte del Start Symbol y se intenta llegar al programa.
  - Parsing Recursivo Descendente
  - LL(1) (Left to Left 1)
- **Parsing Ascendente (Buttom Up):** En el que se parte del programa y se intenta llegar al Start Symbol.
  - SLR (Simple Left to Right)
  - LR(1) (Left to Right 1)
  - LALR

Parsing Descendente (Top Down)

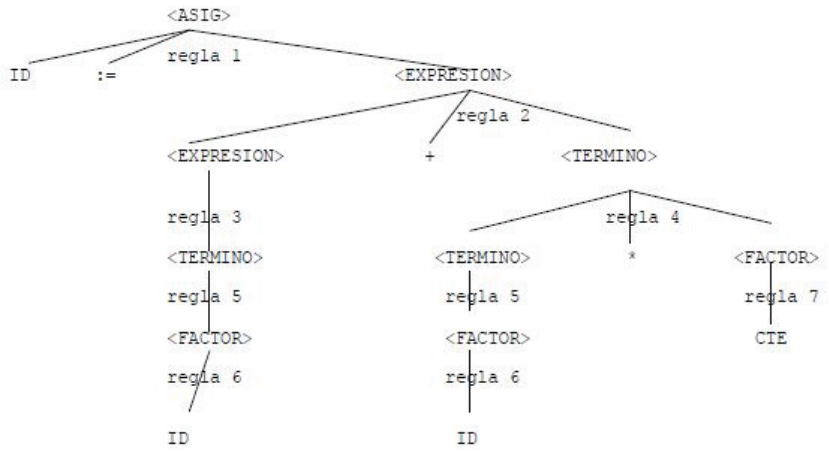
Es el proceso de Análisis Sintáctico en el que se parte del símbolo distinguido hasta llegar al programa de usuario. Se dice que este proceso es LL:
- La primer L significa "Left" porque lee cada contenido de la regla de izquierda a derecha.

- La segunda L significa también "Left" porque aplica las reglas buscando el macheo con la parte izquierda, es decir con el elemento que está siendo definido en una regla.

Ejemplo:

```
1. <ASIG>→ id := <EXPRESION>
2. <EXPRESION>→<EXPRESION> + <TERMINO>
3. <EXPRESION>→<TERMINO>
4. <TERMINO>→<TERMINO> * <FACTOR>
5. <TERMINO>→<FACTOR>
6. <FACTOR>→ id
7. <FACTOR>→ cte
```

Para resolver la siguiente asignación: `ID := ID + ID * CTE`
Hacemos el árbol de parsing:



Tira de reglas aplicadas: 1, 2, 3, 5, 6, 4, 5, 6, 7.

Método "Recursivo Descendente"

Este es el primer método que utilizó la construcción de un árbol descendente. A partir del símbolo distinguido se van aplicando las reglas sobre el No Terminal que se encuentra más a la izquierda.
Ejemplo:
Suponemos las mismas reglas que en el ejemplo anterior, e intentamos resolver la misma asignación:

`ID := ID + ID * CTE`

Entonces:

1	Ponemos el símbolo distinguido	<ASIG>
2	Usamos **regla 1** con el no Terminal <ASIG>	ID := <EXPRESION>
3	Aplicamos la **regla 2** con <EXPRESION> ya que es el primer no terminal desde la izquierda	ID := <EXPRESION> + <TERMINO>
4	Aplicamos la **regla 2** con <EXPRESION> nuevamente ya que es el primer no terminal desde la izquierda	ID := <EXPRESION> + <TERMINO> + <TERMINO>
5	Aplicamos la **regla 2** con <EXPRESION> nuevamente ya que es el primer no terminal desde la izquierda	ID := <EXPRESION> + <TERMINO> + <TERMINO> + <TERMINO>

¡Acá hay un problema! Como el algoritmo lo que hace es aplicar una regla con el no terminal que se encuentre más a la izquierda, esto entra en un bucle infinito. Con esto llegamos a la conclusión de que las gramáticas para algoritmos de parsing descendentes no pueden ser recursivas a izquierda.
Debido a esto, este algoritmo para ser usado debe tener una gramática con BNF recursiva a derecha como la siguiente:

```
1. <ASIG>→ id := <EXPRESION>
2. <EXPRESION>→<TERMINO> + <EXPRESION>
3. <EXPRESION>→<TERMINO>
4. <TERMINO>→<FACTOR> * <TERMINO>
5. <TERMINO>→<FACTOR>
6. <FACTOR>→ id
7. <FACTOR>→ cte
```

Entonces:

1	Ponemos el símbolo distinguido	<ASIG>
2	Usamos **regla 1** con el no Terminal <ASIG>	ID := <EXPRESION>
3	Aplicamos la **regla 2** con <EXPRESION>	ID := <TERMINO> + <EXPRESION>
4	Aplicamos la **regla 4** con <TERMINO> ya que es el primer no terminal desde la izquierda	ID := <FACTOR> * <TERMINO> + <EXPRESION>
5	Luego aplica la **regla 6** para tener la primer variable	ID := ID * <TERMINO> + <EXPRESION>
6	Pero nos encontramos con un problema, el símbolo "*" no se encuentra en nuestro código original así que tenemos que hacer un "Back Tracking" que es volver atrás sobre las reglas aplicadas para probar otro camino. Así que **deshacemos los efectos de la regla 6 y la 4** que hemos realizado y pasamos a hacer la **regla 5**.	D := <FACTOR> + <EXPRESION>
7	Ahora sí, podemos aplicar la **regla 6** para <FACTOR>	ID := ID + <EXPRESION>
8	Se va pareciendo al código original, ahora aplicamos la **regla 2** con <EXPRESION>	ID := ID + <TERMINO> + <EXPRESION>
9	Pero si seguimos con esto nos damos cuenta que el segundo símbolo "+" nos va a provocar un "Back Tracking" así que tenemos que aplicar la **regla 3** en el anterior paso (**deshacemos la regla 2**)	ID := ID + <TERMINO>
10	Aplicamos la **regla 4** con <TERMINO>	ID := ID + <FACTOR> * <TERMINO>
11	Aplicamos la **regla 6** con <FACTOR>	ID := ID + ID * <TERMINO>
12	Ya casi está, aplicamos la **regla 4** con <TERMINO>	ID := ID + ID * <FACTOR> * <TERMINO>
13	Esto nos va a producir otro "Back Tracking" en un paso posterior así que **deshacemos la regla 4** y apliquemos la **regla 5**	ID := ID + ID * <FACTOR>
14	Y por **regla 7** terminamos	ID := ID + ID * CTE

Finalmente aplicamos las siguientes reglas: 1, 2, 5, 6, 3, 4, 6, 5, 7.

Este algoritmo es muy lento, hace muchos "Back Tracking" y tiene mensajes de error complicados.

Método "LL(1)" o "Predictivo Descendente"

Para no tener "Back Tracking" se debe factorizar la gramática.El proceso de factorizar una gramática es obtener cada regla que pueda ser "ambigua" y obtener el factor común de todas ellas para generar un resultado.
Por ejemplo:

Teniendo:

```
E → T + E
E → T
```

Podemos factorizar a:

```
E → T R
R → + E
R → [vacío]
```

Es decir, una expresión es un término seguido de “algo”, ese “algo” puede ser un “+ expresión” o simplemente nada. Con esta gramática y el algoritmo anterior, se obtiene un Analizador Sintáctico que no hace “Back Tracking”. A este método se lo denomina “Método LL(1) o Predictivo Descendente”.

Entonces, la gramática anterior la podemos factorizar para que quede de la siguiente forma:

1. <ASIG>→ id := <EXPRESION>
2. <EXPRESION>→<TERMINO><R>
3. <R>→ + <EXPRESION>
4. <R>→ [vacío]
5. <TERMINO>→<FACTOR><Q>
6. <Q>→ * <TERMINO>
7. <Q>→ [vacío]
8. <FACTOR>→ id
9. <FACTOR>→ cte

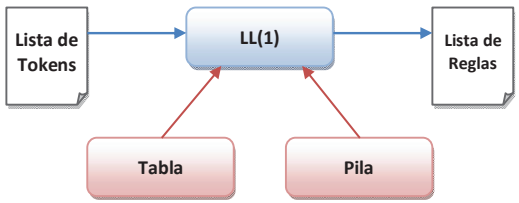
Para resolver la siguiente asignación: ID := ID + ID * CTE

1	Ponemos el símbolo distinguido	<ASIG>
2	Usamos **regla 1** con el no Terminal <ASIG>	ID := <EXPRESION>
3	Aplicamos la **regla 2** con <EXPRESION>	ID := <TERMINO> + <R>
4	Uso **regla 5** con <TERMINO>	ID := <FACTOR><Q><R>
5	Por la **regla 8**, <FACTOR> es un ID	ID := ID <Q><R>
6	Ahora debemos aplicar una regla para <Q>, tenemos dos posibles pero sabemos que la regla 6 no puede ser aplicada ya que el símbolo “*” de la regla entra en conflicto con el símbolo “+” de nuestro código. Por lo tanto usamos la **regla 7**, haciendo desaparecer a <Q>	ID := ID <R>
7	Usamos la **regla 3** con <R> sabiendo que no se puede aplicar la 4 porque tenemos un símbolo “+” en nuestro código original	ID := ID + <EXPRESION>
8	Usamos **regla 2** con <EXPRESION>	ID := ID + <TERMINO><R>
9	Aplicamos la **regla 5** con <TERMINO>	ID := ID + <FACTOR><Q><R>
10	Aplicamos la **regla 8** con <FACTOR>	ID := ID + ID <Q><R>
11	Para <Q> usamos la **regla 6**, ya que tiene el símbolo “*” tal cual se utiliza en el código original	ID := ID + ID * <TERMINO><R>
12	**Regla 5** para <TERMINO>	ID := ID + ID * <FACTOR><Q><R>
13	**Regla 9** para <FACTOR>	ID := ID + ID * CTE <Q><R>
14	**Regla 7** para <Q>	ID := ID + ID * CTE <R>
15	**Regla 4** para <R>	ID := ID + ID * CTE

Finalmente aplicamos las siguientes reglas: 1, 2, 5, 8, 7, 3, 2, 5, 8, 6, 5, 9, 7, 4

Implementación del Método “LL(1)” o “Predictivo Descendente”

La implementación de este método es:



Como entrada recibe la lista de tokens generada por le analizador léxico, utiliza una tabla y una pila para el proceso y tiene como salida una lista de reglas que fueron aplicadas para construir el árbol de parsing. Para explicar en qué consisten ambas estructuras utilizadas por el algoritmo debemos conocer algunos conceptos.

Seguimos trabajando con esta gramática:

1. <ASIG>→ id := <EXPRESION>
2. <EXPRESION>→<TERMINO><R>
3. <R>→ + <EXPRESION>
4. <R>→ [vacío]
5. <TERMINO>→<FACTOR><Q>
6. <Q>→ * <TERMINO>
7. <Q>→ [vacío]
8. <FACTOR>→ id
9. <FACTOR>→ cte

Los Primeros

Se define como “primero” de un No Terminal dado, todo aquel Terminal que aparece en primer lugar dentro de algunas de sus reglas que lo definen.

Obtenemos los primeros de los elementos de la gramática:

1. Por regla 1 vemos que en la asignación lo primero que aparece es un “id”.
   `Prim(ASIG) = { id }`
2. Como EXPRESION tiene como primer elemento a TERMINO, lo que esté primero en TERMINO será su primero.
   `Prim(EXPRESION) = Prim(TERMINO) = { id, cte }`
3. Para R, el primero es un signo “+” ya que el [vacío] no se tiene en cuenta.
   `Prim(R) = { + }`
4. Como TERMINO tiene como primer elemento a FACTOR, lo que esté primero en FACTOR será su primero.
   `Prim(TERMINO) = Prim(FACTOR) = { id, cte }`
5. Para q, el primero es un signo “*” ya que el [vacío] no se tiene en cuenta.
   `Prim(Q) = { * }`
6. En ambas reglas de FACTOR lo que aparece primero son el “id” y la “cte”.
   `Prim(FACTOR) = { id, cte }`

Nota: Si el primer No Terminal de un No Terminal tiene una regla que se hace [vacío] debo obtener los primeros del siguiente.

Ejemplo:

E → T R
T → [vacío]

Entonces:

`Prim(E) = Prim(R)`

Los Siguientes

Se define como “siguiente” de un No Terminal dado, todo aquel Terminal que aparece a continuación pero que no es parte de la definición del No Terminal en cuestión.

Obtenemos los siguientes de los elementos de la gramática:

1. Se dice que para el símbolo distinguido se le da como elemento siguiente al signo $ que representa el final de mi cadena de datos a analizar.
   `Sgt(ASIG) = { $ }`

2. La EXPRESION tiene por un lado la regla 1 por la cual, lo que se encuentre al final de ASIG seguirá al final de EXPRESION. Por otro lado, por la regla 2 vemos que lo que haya al final de R será lo mismo que hay al final de EXPRESION.
   `Sgt(EXPRESION) = Sgt(ASIG) U Sgt(R) = { $ }`
3. En la regla 3 vemos que lo que lo que hay al final de EXPRESION es lo que tiene R al final.
   `Sgt(R) = Sgt(EXPRESION) = { $ }`
4. Por regla 2, lo que viene después de TERMINO es lo primero de R. Pero si R es vacío, lo que viene es lo último de EXPRESION.
   `Sgt(TERMINO) = Prim(R) U Sgt(EXPRESION) = { +, $ }`
5. Por regla 5 es el siguiente de TERMINO
   `Sgt(Q) = Sgt(TERMINO) = { +, $ }`
6. Por regla 5, después de FACTOR está el primero de Q. Por regla 6, después de FACTOR viene el siguiente de Q. Si Q es vacío en la regla 5, lo que viene es el siguiente de TERMINO
   `Sgt(FACTOR) = Prim(Q) U Sgt(Q) U Sgt(TERMINO) = { *, +, $ }`

Tabla que utiliza el Método LL(1)

A partir de los conceptos vistos anteriormente, se arma la siguiente tabla:

	id	:=	+	*	cte	$
ASIG	Id := <EXPRESION>					
EXPRESION	<TERMINO><R>				<TERMINO><R>	
R			+ <EXPRESION>			[vacío]
TERMINO	<FACTOR><Q>				<FACTOR><Q>	
Q			[vacío]	* <TERMINO>		[vacío]
FACTOR	id				cte	

Para construir la tabla se toma cada regla y se hace el siguiente procedimiento, considerando cada una de las reglas de la forma: "α→β", donde a cada uno de los β se lo denomina Forma Sentencial.

- Si β ≠ [vacío] → Se agrega a β en la celda [α, PRIM(α)]
- Si β = [vacío] → Se agrega a β en la celda [α, SGT(α)]

Ejecución del Algoritmo

Vamos a resolver la siguiente asignación: `ID := ID + ID * CTE`

PILA	EXPLICACIÓN	LO QUE FALTA DETECTAR
ASIG	Puse el símbolo distinguido	id := id + id * cte $
EXPRESION := id	Uso la regla 1 metiéndola en la Pila (siempre los elementos van al revés)	id := id + id * cte $
EXPRESION :=	Saco el "id" de la pila ya que fue reconocido	:= id + id * cte $
EXPRESION	Saco el ":=" de la pila ya que fue reconocido	id + id * cte $
R TERMINO	Uso la regla 2	id + id * cte $
R Q FACTOR	Uso la regla 5	id + id * cte $
R Q id	Uso la regla 8	id + id * cte $
R Q	Saco el "id" de la pila ya que fue reconocido	+ id * cte $
R	Uso la regla 7	+ id * cte $
EXPRESION +	Uso la regla 3	+ id * cte $
EXPRESION	Saco el "+" de la pila ya que fue reconocido	id * cte $
R TERMINO	Uso la regla 2	id * cte $
R Q FACTOR	Uso la regla 5	id * cte $

R Q id	Uso la regla 8	id * cte $
R Q	Saco el "id" de la pila ya que fue reconocido	* cte $
R TERMINO *	Uso la regla 6	* cte $
R TERMINO	Saco el "*" de la pila ya que fue reconocido	cte $
R Q FACTOR	Uso la regla 5	cte $
R Q cte	Uso la regla 9	cte $
R Q	Saco la "cte" de la pila ya que fue reconocido	$
R	Uso la regla 7	$
$	Uso la regla 4 y reconozco el "$"	

Finalmente aplicamos las siguientes reglas: 1, 2, 5, 8, 7, 3, 2, 5, 8, 6, 5, 9, 7, 4

Parsing Ascendente (Buttom Up)

Es el proceso de Análisis Sintáctico en el que se parte del programa de usuario y se llega al símbolo distinguido. Se dice que este proceso es LR:

- La L significa "Left" porque lee cada contenido de la regla de izquierda a derecha.
- La R significa "Right" porque aplica las reglas buscando el macheo con la parte derecha.

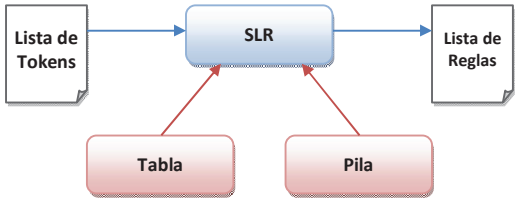
Los métodos son:

- SLR
- LR (1)
- LALR

Con estos métodos ya no es necesario que las gramáticas estén factorizadas y es indiferente que sean recursivas a derecha o a izquierda. Lo único necesario es que no sea una gramática ambigua.

Método "SLR"

La implementación de este método es:



Tenemos la siguiente gramática:

```
0. <A'>→<A>
1. <A>→ id := <E>
2. <E>→<E> + <T>
3. <E>→<T>
4. <T>→<T> * <F>
5. <T>→<F>
6. <F>→ id
7. <F>→ cte
```

Como se puede apreciar, la gramática esta aumentada, eso quiere decir que existe la regla cero que engloba al que a priori sería nuestro símbolo distinguido. Ahora el distinguido es <A'>.

Algoritmo de Construcción de Tabla

Antes de comenzar con el algoritmo, cabe explicar una notación nueva que usaremos. Al analizar una regla, se le va a incluir un punto (.) en un lugar determinado, de manera que lo que está delante del punto se considera ya analizado, y lo que está después del punto por analizar.

Ejemplos:
Recién agarro la regla, debo buscar información sobre <T>:
<E>→. <E> + <T>
Ya analicé <E>, ahora busco información sobre + <T>:
<E>→<E> . + <T>
Terminé de aplicar la regla:
<E>→<E> + <T> .

El algoritmo para generar la tabla consta de ir navegando por las reglas e ir generando estados. Veámoslo en forma práctica para la gramática anteriormente mostrada:

Estado 0:
Comenzamos generando este estado con la regla cero:
<A'>→. <A>
Observe que el punto adelante indica que aún no hemos visto el resto de la regla. A continuación debemos definir qué elementos válidos son los que pueden llegar a venir a continuación del punto. Como el punto está delante de un No Terminal, debemos aplicar una regla para resolverlo, a medida que hacemos esto generamos otro estado.
Entonces, teniendo el No Terminal <A>, debemos ver que reglas lo definen:
1. <A>→id := <E>
Bueno, sabiendo esto, tenemos que probar que pasa cuando estando en estado 0, nos llega un <A> (que es el No Terminal buscado) y un "id" que es el primer elemento que aparece en nuestra única regla posible. Para cada posibilidad vamos a generar un estado.
Resumiendo, los elementos válidos para el estado 0 que pueden venir a continuación son: <A>, "id".

Estado 1 (0, A):
Generamos un estado nuevo que surge de recibir el No Terminal <A> estando en el estado 0 (de ahí las dos cosas que están contenidas en el paréntesis). Esto significa que hemos tenido en cuenta el <A> que esperábamos en el estado 0, dejándonos:
<A'>→<A> .
Observemos que el punto avanzo porque ya atendimos la llegada del <A>. Con lo cual, cuando el punto se encuentra al final de todo, damos por terminado el análisis de este estado.

Estado 2 (0, id):
Este es el otro caso en el que estando en estado 0 nos llega un "id", para él generamos un nuevo estado que llamamos 2. Ahora tenemos:
<A>→id . := <E>
Estamos dentro de la regla 1 porque este estado surgió de la posibilidad de aplicar dicha regla. Por lo tanto, lo que hace es comenzar con el avance de la regla. Por eso, y luego de haber consumido el "id" ubicamos el punto por detrás del ":=".
En este estado el elemento siguiente es el Terminal ":=". Por este motivo, el único elemento que debe venir a continuación es ese.

Estado 3 (2, :=):
Avanzamos por la regla 1, tenemos:
<A>→id := . <E>
Bueno, ahora volvemos a tener un No Terminal delante del punto, esta vez es <E>. Entonces buscamos las reglas que definen a este No Terminal:

2. <E>→. <E> + <T>
3. <E>→. <T>
4. <T>→. < T> * <F>
5. <T>→. <F>
6. <F>→. id
7. <F>→. cte

Además de las reglas 2 y 3 que definen a <E> se incluyeron las reglas 4 y 5 porque una <E> es un <T> según regla 3. Por el mismo motivo se agregaron las reglas 6 y 7 a través de <F>.
Recuerde que <E> se incluye por dos motivos en la lista, primero porque es el elemento No Terminal que está delante del punto y segundo porque existe una regla (la 2) en la que es el primer elemento en aparecer dentro de la definición.
Elementos válidos a continuación: <E>, <T>, <F>, id, cte.

Estado 4 (3, E):
Acá hay dos cuestiones, la primera es tomar a <E> por haber sido el elemento que en el estado 3 estaba detrás del punto, para lo cual podemos medir el avance:
<A>→id := <E> .
Por otro lado, tomamos a <E> como el primer elemento de la regla 2 que era la primera posibilidad del estado anterior. Según esto, debemos medir el avance de dicha regla para lo cual estamos así:
<E>→<E>. + <T>
Bueno, ahora debemos definir qué elementos vienen a continuación. Si miramos el primer avance, vemos que el punto está al final de la regla con lo cual ese avance termina ahí. Ahora vemos el segundo avance y se observa que hay un Terminal "+" a continuación.
Elementos válidos a continuación: +

Estado 5 (3, T):
En este caso ocurre lo mismo, tenemos dos avances:
<E>→<T>.
<T>→<T>. * <F>
Esto se debe a que en el estado anterior hay dos reglas que comienzan con <T> en su cuerpo de definición. Tal como vimos, el primer avance se descarta por haber finalizado y con el otro esperamos a "*".
Elementos válidos a continuación: *

Estado 6 (3, F):
En este estado, el avance es:
<T>→<F>.
Y acá terminamos con este estado.

Estado 7 (3, id):
En este estado, el avance es:
<F>→id .
Y acá terminamos con este estado.

Estado 8 (3, cte):
En este estado, el avance es:
<F>→cte .
Y acá terminamos con este estado.

Estado 9 (4, +):
En este estado, el avance es:
<E>→<E> + . <T>
Ahora nos encontramos con el No Terminal <T>, veamos que reglas puedo aplicar:
4. <T>→. < T> * <F>
5. <T>→. <F>
6. <F>→. id
7. <F>→. cte

Elementos válidos a continuación: <T>, <F>, id, cte.

Estado 10 (5, *):

En este estado, el avance es:

```
<T>→<T> * . <F>
```

Se viene el <F>, veamos que reglas puedo aplicar:

```
6. <F>→. id
7. <F>→. cte
```

Recordemos que además del "id" y la "cte", un elemento que se espera en este estado es <F> ya que es el No Terminal que está a continuación del punto.
Elementos válidos a continuación: <F>, id, cte.

Estado 11 (9, T):

Tenemos:

```
<E>→<E> + <T> .
<T>→<T> . * <F>
```

Como para este No Terminal hay una regla además de ser el esperado, tiene dos avances. En el primero se termina la regla y en el segundo se espera un "*" a continuación.
Elementos válidos a continuación: *

~~Estado 12 (9, F):~~

Este estado finalmente no es tenido en cuenta. Esto es así porque de aplicar <F> al estado 9, el avance quedaría así:

```
<T>→<F> .
```

Observe que ese avance es exactamente el mismo que el que tiene el estado 6, con lo cual no se tiene en cuenta. Lo que si vamos a tener en cuenta es que el par (9, F) va a estar asociado con el estado 6:

Estado 6 (9, F)

~~Estado 12 (9, id):~~

Pasa lo mismo que en el anterior pero con el estado 7.

Estado 7 (9, id)

~~Estado 12 (9, cte):~~

Pasa lo mismo que en el anterior pero con el estado 8.

Estado 8 (9, cte)

Estado 12 (10, F):

El avance es:

```
<T>→<T> * <F>.
```

Terminamos con este estado.

~~Estado 13 (10, id):~~

Pasa lo mismo que en el anterior pero con el estado 7.

Estado 7 (10, id)

~~Estado 13 (10, cte):~~

Pasa lo mismo que en el anterior pero con el estado 8.

Estado 8 (10, cte)

~~Estado 13 (11, *):~~

Pasa lo mismo que en el anterior pero con el estado 10.

Estado 10 (11, *)

Listo, a este punto no tenemos abierto ningún estado de los que generamos con lo cual damos por finalizado el algoritmo de generación de estados.

La tabla está compuesta por tantas filas como estados hayamos generado (en este ejemplo son 13 estados) y en cuanto a las columnas está dividida en dos, el primer grupo para los No Terminales y el segundo para los Terminales. La parte sombreada de celeste se le llama "Acción" y la parte verde "Goto".

ESTADO	id	:=	+	*	cte	$	A	E	T	F
0										
1										
2										
3										
4										
5										
6										
7										
8										
9										
10										
11										
12										

Completar la parte de Goto:

Para llenar la parte de "Goto", vamos a detectar aquellos estados en los que se llega cuando recibe un Terminal.

Estos son:

- ***Estado 1 (0, A)***
- ***Estado 4 (3, E)***
- ***Estado 5 (3, T)***
- ***Estado 6 (3, F)***
- ***Estado 11 (9, T)***
- ***Estado 6 (9, F)***
- ***Estado 12 (10, F)***

Observe que los elegidos son cada uno de los pares cuyo segundo elemento es un No Terminal. Con cada uno se debe ubicar la fila del estado original, la columna del No Terminal y dentro de la celda escribir el estado destino. Por ejemplo, para la fila 0 y columna A el valor será 1.

Esta sección que se llama "Goto" básicamente lo que nos brinda es la posibilidad de saber a qué estado debemos ir cuando el algoritmo del analizador sintáctico SLR reconoce un No Terminal, dependiendo del estado en que se encuentre.
Por ejemplo, si se encuentra en estado 3 y reconoce una <EXPRESION> debe ir al estado 4.Observe que el avance del estado 4 era el siguiente:

```
<A>→id := <E> .
```

Lo que significa que hemos reconocido y consumido la <E> que nos había llegado.

Completar la parte de Acciones:

Para llenar la sección de "Acciones". Debemos saber que existen dos tipos de acciones:

- **Desplazar**: Es el acto de consumir el carácter siguiente que forma parte de la lectura de la entrada del usuario de izquierda a derecha. Por ejemplo, cuando se tiene "id := ... " estando parado entre ambos caracteres y se solicita avanzar consumiendo el ":=".
- **Reducir**: Es el acto de reconocer a un grupo de elementos y definirlos como el elemento que los engloba según la regla apropiada. Por ejemplo, cuando se reconoce a <E> + <T> como una<E>.

Para llenar los casilleros correspondientes a los desplazamientos vamos a detectar aquellos estados en los que se llega cuando recibe un No Terminal.

Estos son:

- ***Estado 2 (0, id)***
- ***Estado 3 (2, :=)***
- ***Estado 7 (3, id)***
- ***Estado 8 (3, cte)***
- ***Estado 9 (4, +)***
- ***Estado 10 (5, *)***
- ***Estado 7 (9, id)***
- ***Estado 8 (9, cte).***
- ***Estado 7 (10, id).***
- ***Estado 8 (10, cte).***
- ***Estado 10 (11, *).***

Observe que los elegidos son cada uno de los pares cuyo segundo elemento es un Terminal. Con cada uno se debeubicar la fila del estado original, la columna del Terminal y dentro de la celda escribir el estado destino junto a una"D" que significa desplazamiento. Por ejemplo, para la fila 0 y columna "id" el valor será "D2".

Para llenar los casilleros correspondientes a las reducciones primero debemos obtener aquellos estados que tienen al menos una regla con el punto al final de todo, es decir, cuando han completado el avance.
Estos son:

- ***Estado 1 (0, A)*** ***<A'>→<A> .***
- ***Estado 4 (3, E)*** ***<A>→ id := <E> .***
- ***Estado 5 (3, T)*** ***<E>→<T> .***
- ***Estado 6 (3, F)*** ***<T>→<F> .***
- ***Estado 7 (3, id)*** ***<F>→id .***
- ***Estado 8 (3, cte)*** ***<F>→cte .***
- ***Estado 11 (9, T)*** ***<E>→<E> + <T> .***
- ***Estado 6 (9, F)*** ***<T>→<F> .***
- ***Estado 7 (9, id)*** ***<F>→id .***
- ***Estado 8 (9, cte)*** ***<F>→cte .***
- ***Estado 12 (10, F)*** ***<T>→<T> * <F> .***
- ***Estado 7 (10, id)*** ***<F>→id .***
- ***Estado 8 (10, cte)*** ***<F>→ cte .***

Antes de explicar cómo se completa la tabla, hay que obtener los "Siguientes" de cada elemento No Terminal:

```
Sgt(A') = { $ }
Sgt(A) = Sgt(A') = { $ }
Sgt(E) = Sgt(A) U { + } = { $, + }
Sgt(T) = Sgt(E) U { * } = { $, +, * }
Sgt(F) = Sgt(T) = { $, +, * }
```

Bueno, por ejemplo, para el estado 1 que obtuve en la lista anterior estoy aplicando la regla 0 que define al No Terminal <ASIG'>, tomamos la fila 1 y las columnas que representen los "Siguientes" de ese elemento (en este caso solo "$") y escribo en la celda un "R0" ya que la regla es la 0.
Para el estado 4, tomamos los "Siguientes" de <ASIG> que también es "$" y escribimos un "R1" ya que la regla que se aplica en este estado es la 1.

Por último vamos a hacer un retoque. Es lógico pensar que si aplicamos la regla 0 en algún momento del algoritmo podemos decir que el programa está bien estructurado porque estaríamos llegando al símbolo distinguido. Con lo cual ponemos un "ok" en la aplicación de la regla. La tabla queda de la siguiente manera:

ESTADO	id	:=	+	*	cte	$	A	E	T	F
0	D2						1			
1						OK				
2		D3								
3	D7				D8			4	5	6
4			D9			R1				
5			R3	D10		R3				
6			R5	R5		R5				
7			R6	R6		R6				
8			R7	R7		R7				
9	D7				D8				11	6
10	D7				D8					12
11			R2	D10		R2				
12			R4	R4		R4				

Y con esto finalizamos la construcción de la tabla.

Ejecución del Algoritmo

Vamos a resolver la siguiente asignación:

```
ID := ID + ID * CTE
```

Lo que vamos a tener en la pila son los estados en los que estaremos a cada momento. También vamos a tener unsímbolo que será nuestro elemento activo y en la última columna tendremos nuestra entrada que iremos consumiendohasta finalizar.

PILA	SÍMBOLO	EXPLICACIÓN	LO QUE FALTA DETECTAR
0		Iniciamos en el estado 0	id := id + id * cte $
0 2	id	Recibimos el id, buscamos la celda [0, id] yencontramos D2. Con lo cual consumimos el id yquedamos en estado 2.	:= id + id * cte $
0 2 3	id :=	Recibimos el :=, con [2, :=] = D3, desplazamosy quedamos en estado 3.	id + id * cte $
0 2 3 7	id := id	Recibimos un Nuevo id, con [3, id] = D7,desplazamos y nos vamos al estado 7.	+ id * cte $
0 2 3 6	id := F	Ahora encontramos un +, y en él hay un reducirporque [7, +] = R6. La regla 6 indica F → id,con lo cual tomo el último id y lo transformo aF. Además debo quitar un elemento de estado dela pila, quedándome con el estado anterior quees 3. Luego, utilizo la tabla de GOTO con elestado actual y el elemento No Terminal con elque se ha reducido, esto da [3, F] = 6, y esteúltimo es el nuevo estado. Observe que elsímbolo + no ha sido consumido porque no hubodesplazamiento.	+ id * cte $
0 2 3 5	id := T	Vuelvo a recibir el +, tenemos [6, +] = R5.Reducimos por regla 5, que es T → F, quitamosel estado 6 y hacemos [3, T] = 5, este últimoes nuestro nuevo estado.	+ id * cte $
0 2 3 4	id := E	Vuelvo a recibir el +, tenemos [5, +] = R3.Reducimos por regla 3, que es E → T, quitamosel estado 5 y hacemos [3, E] = 4, este últimoes nuestro nuevo estado.	+ id * cte $
0 2 3 4 9	id := E +	Vuelvo a recibir el +, con [4, +] = D9,desplazamos el +.	id * cte $
0 2 3 4 9 7	id := E + id	Recibo un id, con [9, id] = D7.	* cte $
0 2 3 4 9 6	id := E + F	Recibo un *, con [7, *] = R6. La regla 6 es F→ id. Quitamos el estado 7 y hacemos [9, F] =6.	* cte $
0 2 3 4 9 11	id := E + T	Recibo un *, con [6, *] = R5. La regla 5 es T→ F. Quitamos el estado 6 y hacemos [9, T] =11.	* cte $
0 2 3 4 9 11 10	id := E + T *	Recibimos un *, con [11, *] = D10, consumimosel *.	cte $

0 2 3 4 9 11 10 8	id := E + T * cte	Recibimos la cte, con [10, cte] = D8,desplazamos la cte.	$
0 2 3 4 9 11 10 12	id := E + T * F	Nos llega el símbolo final de la gramática, con[8, $] = R7, la regla 7 es F → cte, y [10, F]= 12.	$
0 2 3 4 9 11	id := E + T	Recibí el $, con [12, $] = R4, regla 4 es T →T * F, la utilizo. Al momento de quitarestados, no vamos a quitar uno solo sino losúltimos 3 estados ya que la definición de laregla tiene 3 elementos (T * F), con lo cualquedamos en el estado 9. Hacemos [9, T] = 11 yese es nuestro nuevo estado	$
0 2 3 4	id := E	Recibimos nuevamente el $, con [11, $] = R2,aplicamos regla 2 que es E → E + T. Quitamos 3estados quedándonos con el estado 3, y hacemos[3, E] = 4.	$
0 1	A	Recibimos $, con [4, $] = R1, usamos regla 1que es A → id := E, reducimos y quitamos los 3estados quedándonos con el 0. Con [0, A] = 1.	$
		Recibimos $, con [1, $] = OK, con lo cualaplicamos regla 0 (A' → A) Y finalizamosconcluyendo que la sentencia no tiene erroressintácticos.	$

Finalmente aplicamos las siguientes reglas: 6, 5, 3, 6, 5, 7, 4, 2, 1, 0.

Generación de Código Intermedio

Ahora veremos el proceso de generar código. Este proceso recibe como entrada una lista de reglas que el Analizador Sintáctico utilizo para procesar la entrada de tokens.

Vamos a usar esta gramática de ejemplo:

1. <ASIG>→ id := <EXPRESION>
2. <EXPRESION>→<EXPRESION> + <TERMINO>
3. <EXPRESION>→<EXPRESION>- <TERMINO>
4. <EXPRESION>→<TERMINO>
5. <TERMINO>→<TERMINO> * <FACTOR>
6. <TERMINO>→<TERMINO> / <FACTOR>
7. <TERMINO>→<FACTOR>
8. <FACTOR>→ id
9. <FACTOR>→ cte

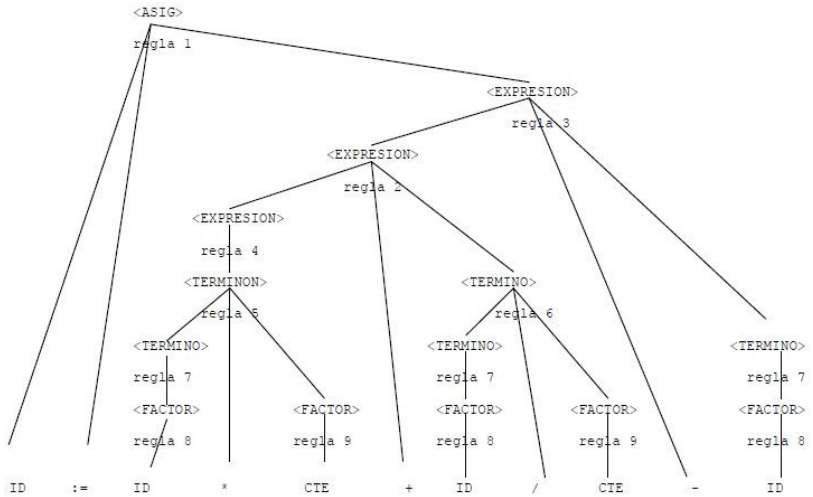
Supongamos que tenemos código:

```
precio := costo * 1.4 + transporte / 20 – bonific
```

La sentencia puede verse como:

```
id := id * cte + id / cte – id
```

Si hacemos el árbol de parsing:



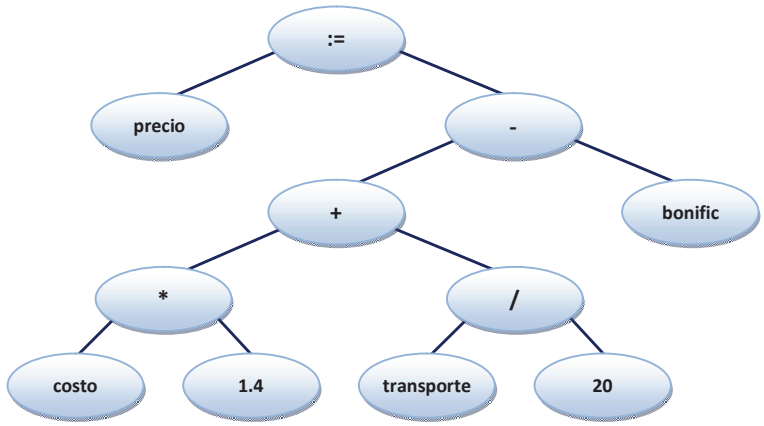
El resultado de ese proceso es la tira de reglas, que es lo que arroja el Analizador Sintáctico, como se dijo anteriormente, pero además de esto, las reglas 8 y 9 vienen acompañadas por el índice de la Tabla de Símbolos a la que apuntan:

8	7	9	5	4	8	7	9	6	2	8	7	3	1
↓		↓			↓		↓			↓			
costo		*1.4*			*transporte*		*20*			*bonific*			

Ahora si, esta es la entrada tal cual la toma el Generador de Código junto a la Tabla de Símbolos.

Árbol Sintáctico

El árbol sintáctico es una estructura similar al árbol de parsing como el que hicimos anteriormente en la que se eliminan los No Terminales. Se muestra a continuación:



Cada nodo es una operación entre sus dos hijos. Por ejemplo, el nodo "*" es una multiplicación entre "costo" y "1.4", el nodo "/" es una división entre "trasporte" y "20", el resultado de ambos es una suma dentro del nodo "+" y así sucesivamente. En la práctica, los nodos que apuntan a una constante o a una variable tiene el índice a la Tabla de Símbolos.

Si el árbol es recorrido en orden simétrico se obtiene el mismo programa que se está compilando. La forma de recorrerlo por este método es empezar con el nodo padre, en cada nodo primero se resuelve el primer hijo, luego se toma el nodo padre y luego se resuelve el segundo hijo. Por ejemplo, comienzo con ":=" tomando a "precio", luego tomo el ":=" y luego resuelvo todo lo que hay debajo de "-" (el resultado será "costo * 1.4 + transporte / 20" si continuamos con el algoritmo). Finalmente tomamos el "-", luego "bonific" y listo.

Polaca Inversa

Pero existe otra forma de tomar la entrada, esta se llama Polaca Inversa. Con este método, se debe tomar el nodo padre y en cada nodo primero se resuelve el primer hijo, luego el segundo nodo y por último el mismo padre.
Si hacemos esto el resultado será:

```
precio costo 1.4 * transporte 20 / + bonific - :=
```

Para verlo un poco mas claro, se debe tener en cuenta que cada operador toma como operandos los dos que tiene por delante:

- El "*" afecta a "costo" y "1.4".
- El "/" afecta a "transporte" y "20".
- El "+" afecta a "(costo * 1.4)" y "(transporte / 20)".
- El "-" afecta a "((costo * 1.4) +(transporte / 20))" y "bonific".
- El ":=" afecta a "(((costo * 1.4) +(transporte / 20)) - bonific)" y "precio".

Tercetos

Otra forma de representar las operaciones es la llamada Tercetos. Es una agrupación de cada uno de los nodos de manera cada uno tiene los tres elementos utilizando referencias entre las operaciones intermedias:

```
[1]. (*, costo, 1.4)
[2]. (/, transporte, 20)
[3]. (+, [1], [2])
[4]. (-, [3], bonific)
[5]. (:=, precio, [4])
```

Los valores entre corchetes son referencias a otros de los tercetos.

Cuartetos

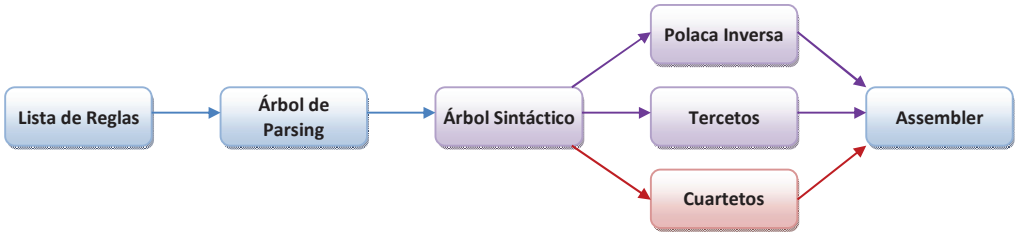
Otra forma parecida a esta es la de Cuartetos. Es igual a la anterior pero con un elemento mas que es el sitio simbólico donde se guarda el resultado de la operación intermedia:

```
[1]. (*, costo, 1.4, aux1)
[2]. (/, transporte, 20, aux2)
[3]. (+, aux1, aux2, aux3)
[4]. (,aux3, bonific, aux4)
[5]. (:=, precio, aux4, …)
```


Generación de Código Final

Generación de Código Assembler

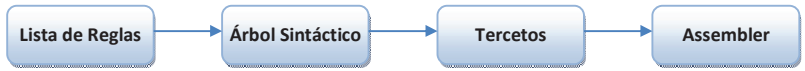
Luego de esta introducción a los métodos, vamos a ver el detalle el proceso de generación de Código Assembler:



En este cuadro está ilustrado el camino original por el que debe pasar la lista de reglas para transformarse en código Assembler. En realidad el Árbol de Parsing no se usa, es un concepto teórico, mientras que se debe utilizar Tercetos o Polaca Inversa ya que el método de Cuartetos es obsoleto (se usaba antiguamente en máquinas con instrucciones de tres operandos).
Pero existen posibilidades de saltarse muchos de los pasos con lo cual hay muchos procesos diferentes de generación. El punto está en elegir que camino tomaremos a la hora de desarrollar nuestro proceso. Esa decisión se va a basar respecto a los objetivos que tengamos, los cual pueden ser:

- Se quiere obtener un compilador rápido y compacto

- Se quiere que nuestro compilador obtenga ejecutables eficientes



- Poco esfuerzo a la hora de desarrollar nuestro compilador

Todas las etapas menos la de Assembler dependen del lenguaje que estamos creando, pero a su vez esta depende de la máquina donde corre los ejecutables que se obtengan como resultado. Con lo cual, la primera opción no es portable, en cambio, las otras opciones solo requieren que se desarrolle nuevamente la etapa de Assembler al momento de transformar nuestro compilador a otra plataforma.
La segunda opción ejecuta las etapas donde se tratan todas las optimizaciones conocidas, por eso genera ejecutables más eficientes (Ver sección *Optimización de Código*).

Traducción de Lista de Reglas a Notación Intermedia

Al realizar la traducción de la lista de reglas a una notación intermedia en particular (árbol sintáctico, tercetos o polaca inversa), por cada una de las reglas que se aplica, se debe aplicar las **Acciones Semánticas** necesarias para poder realizar la traducción.

Utilizaremos la siguiente gramática:

```
1. <A>→ id := <E>
2. <E>→<E> + <T>
3. <E>→<E>- <T>
4. <E>→<T>
5. <T>→<T> * <F>
6. <T>→<T> / <F>
7. <T>→<F>
8. <F>→ id
9. <F>→ cte
```

Lista de Reglas → Árbol Sintáctico

A continuación vamos a explicar la forma de construir el árbol sintáctico en memoria. La idea es generar en memoria una serie de nodos que están interconectados. Para eso, cada uno tiene dos punteros, uno al hijo izquierdo y el otro al hijo derecho. A su vez vamos a tener tantos punteros a nodo como No Terminales exista en nuestra gramática. Estos punteros van a ir siendo desplazados a medida de que se va reconociendo la sentencia a compilar. Para nuestro caso estos serían: Ap, Ep, Tp y Fp.

El algoritmo lo que hace es ejecutar una función asociada a cada regla en el momento de aplicarla. A continuación vamos a explicar cada una:

N°	REGLA	ACCIÓN SEMÁNTICA	EXPLICACIÓN
1	<A>→ id := <E>	Ap = crearNodo(":=", crearHoja(id), Ep)	Se crea el nodo que contiene al signo de asignación y toma como hijo izquierdo una hoja creada conel "id" reconocido y como hijo derecho la expresión reconocida en Ep. El nodo resultante queda apuntado por el puntero Ap.
2	<E>→<E> + <T>	Ep = crearNodo("+", Ep, Tp)	Crea un nuevo nodo para la suma y pone como hijos a la expresión y al término reconocidos.
3	<E>→<E> - <T>	Ep = crearNodo("-", Ep, Tp)	Parecido al anterior pero con el nodo de resta.
4	<E>→<T>	Ep = Tp	Con esta sentencia, el nodo que estaba apuntando Tp (que hasta ese momento era unTermino) ahora pasa a ser apuntado por Ep (pasa a ser una Expresión).
5	<T>→<T> * <F>	Tp = crearNodo("*", Tp, Fp)	Se crea el nodo que contiene a la multiplicación y toma como hijo izquierdo al nodo apuntado en Tp y como hijo derecho al nodo apuntado por Fp. El nodo resultante queda apuntado por elpuntero Tp.
6	<T>→<T> / <F>	Tp = crearNodo("/", Tp, Fp)	Parecido al anterior pero con el nodo de división.
7	<T>→<F>	Tp = Fp	Con esta sentencia, el nodo que estaba apuntando Fp (que hasta ese momento era unFactor) ahora pasa a ser apuntado por Tp (pasa a ser un Termino).
8	<F>→ id	Fp = crearHoja(id)	Esta función crea un nuevo nodo en memoria y le pone el valor pasado como parámetro al contenido.En este caso "id".
9	<F>→ cte	Fp = crearHoja(cte)	Esta función crea un nuevo nodo en memoria y le pone el valor pasado como parámetro al contenido.En este caso "cte".

Para ejecutar el algoritmo, debemos recorrer la lista de reglas e ir ejecutando los pasos según la tabla anterior. Al finalizar, el puntero Ap queda apuntando al comienzo del árbol sintáctico que quedó listo en memoria.

NOTA:
Tomemos como ejemplo el siguiente código:


```
int x, y;
x = y + 2;
```

Sabemos que la primera sentencia, que es de declaración, no va a generar código Assembler. Por lo tanto nos podemos preguntar: ¿es necesario armar un árbol sintáctico que contenga las declaraciones?
La respuesta es que no está mal hacerlo pero no es necesario. Lo más práctico es separar las sentencias ejecutables de las de declaración y dejar que estas últimas se encarguen de generar la Tabla de Símbolos.

Lista de Reglas → Tercetos

Para generar los tercetos a partir de la lista de reglas, se procede de manera similar a la construcción del árbol sintáctico. Se utilizan índices para cada No Terminal que exista en nuestra gramática. Estos índices van a ir siendo actualizados a medida de que se va reconociendo la sentencia a compilar. Para nuestro caso estos serían: Ai, Ei, Ti y Fi.

El algoritmo lo que hace es ejecutar una función asociada a cada regla en el momento de aplicarla. A continuación vamos a explicar cada una:

N°	REGLA	ACCIÓN SEMÁNTICA
1	<A>→ id := <E>	Ai = crearTerceto(":=", id, Ei)
2	<E>→<E> + <T>	Ei = crearTerceto("+", Ei, Ti)
3	<E>→<E> - <T>	Ei = crearTerceto("-", Ei, Ti)
4	<E>→<T>	Ei = Ti
5	<T>→<T> * <F>	Ti = crearTerceto("*", Ti, Fi)
6	<T>→<T> / <F>	Ti = crearTerceto("/", Ti, Fi)
7	<T>→<F>	Ti = Fi
8	<F>→ id	Fi = crearTerceto(id, "_", "_")
9	<F>→ cte	Fi = crearTerceto(cte, "_", "_")

Para ejecutar el algoritmo, debemos recorrer la lista de reglas e ir ejecutando los pasos según la tabla anterior.
El resultado de este procedimiento se suele almacenar en un archivo de texto, en el cual se agrega un nuevo terceto cada vez que se llama a la función "crearTerceto()". Esta función además, va enumerando cada terceto que se va generando en el archivo.

Por ejemplo:
Si se tiene la lista de reglas: **8, 7, 9, 5, 4, 9, 7, 2, 1**

Se generarán los siguientes tercetos:

```
[1]. (id2, _, _)
[2]. (cte1, _, _)
[3]. (*, [1], [2])
[4]. (cte2, _, _)
[5]. (+, [3], [4])
[6]. (:=, id1, [5])
```

También puede optimizarse el algoritmo para generar (sin crear un terceto para cada `id` o `cte`, sino que se guardan en alguna variable hasta que se utilicen en alguna operación).

```
[1]. (*, id2, cte1)
[2]. (+, [1], cte2)
[3]. (:=, id1, [2])
```

Lista de Reglas → Polaca Inversa

Para generar la notación Polaca Inversa a partir de la lista de reglas no es necesario utilizar puntero o índices como en los casos anteriores.

El algoritmo lo que hace es ejecutar una función asociada a cada regla en el momento de aplicarla. A continuación vamos a explicar cada una:

N°	REGLA	ACCIÓN SEMÁNTICA
1	<A>→ id := <E>	generaPolaca(id) generaPolaca(":=")
2	<E>→<E> + <T>	generaPolaca("+")
3	<E>→<E> - <T>	generaPolaca("-")
4	<E>→<T>	
5	<T>→<T> * <F>	generaPolaca("*")
6	<T>→<T> / <F>	generaPolaca("/")
7	<T>→<F>	
8	<F>→ id	generaPolaca(id)
9	<F>→ cte	generaPolaca(cte)

Para ejecutar el algoritmo, debemos recorrer la lista de reglas e ir ejecutando los pasos según la tabla anterior.Existen reglas que no generan términos para la Polaca Inversa, como la 4 y la 7.
El resultado de este procedimiento se suele almacenar en un archivo de texto al igual que para los tercetos. Con cada llamada a la función "generarPolaca()" se agrega un nuevo Terminal al archivo de texto.
Es necesario reemplazar "id" por el nombre que tiene dicho "id" para luego poder referenciarlo más adelante.

Por ejemplo:
Si se tiene la lista de reglas: **8, 7, 9, 5, 4, 9, 7, 2, 1**
Se generarán la siguiente Polaca Inversa:

```
id2  cte1  *  cte2  +  id1  :=
```

NOTA:
Es conveniente aclarar que esta forma de generar la Polaca Inversa es como la genera el YACC, pero que la Polaca Inversa propiamente dicha debería ser de la siguiente manera:

```
id1id2  cte1  *  cte2  +  :=
```

En la cual se cambia la posición del "id" en el cual se realiza la asignación, el cual se agregaría al principio.

Tipos de Datos en construcción del Árbol Sintáctico

El algoritmo explicado anteriormente no tiene en cuenta si las variables y constantes son de diferente tipo de dato. Por ejemplo, "costo" podría ser un float, "bonific" un long, etc. Vamos a ver qué se modifica del algoritmo anterior para ser contemplado este tema.
En primer lugar, en vez de tener solo un puntero por cada No Terminal, ahora tenemos una estructura por cada uno. Esta estructura está compuesta por el puntero y por el tipo de dato.

Entonces serian:

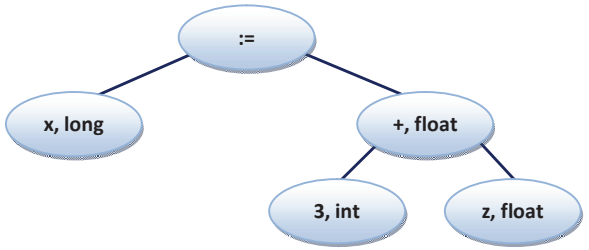
- A { A.p, A.tipo }
- E { E.p, E.tipo }
- T { T.p , T.tipo }
- F { F.p, F.tipo }

Y actualizamos las funciones que se ejecutan al aplicar las reglas, ya que ahora hay que tratar con las estructuras:

N°	REGLA	FUNCIONES
1	<A>→ id := <E>	A.p = crearNodo(":=", crearHoja(id, id.tipo), Ep, id.tipo)
2	<E>→<E> + <T>	E.tipo = fun(E.tipo, T.tipo) E.p = crearNodo("+", Ep, Tp, E.tipo)
3	<E>→<E> - <T>	E.tipo = fun(E.tipo, T.tipo) E.p = crearNodo("-", Ep, Tp, E.tipo)
4	<E>→<T>	E.tipo = T.tipo E.p = Tp
5	<T>→<T> * <F>	T.tipo = fun(T.tipo, F.tipo) T.p = crearNodo("*", Tp, Fp, T.tipo)
6	<T>→<T> / <F>	T.tipo = fun(T.tipo, F.tipo) T.p = crearNodo("/", Tp, Fp, T.tipo)
7	<T>→<F>	T.tipo = F.tipo T.p = Fp
8	<F>→ id	F.tipo = id.tipo F.p = crearHoja(id)
9	<F>→ cte	F.tipo = cte.tipo F.p = crearHoja(cte)

La función llamada "fun" en las reglas que mostramos anteriormente es utilizada para obtener un tipo de dato resultante de operar con los dos tipos de datos pasados como parámetros.

Vamos a ver un ejemplo. Si tenemos: `x := 3 + z`, el árbol sería:

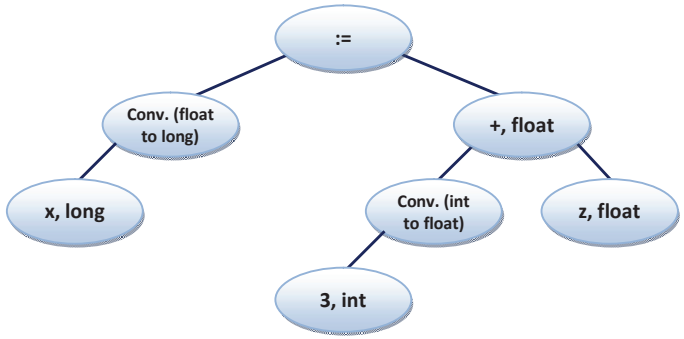


Cada nodo tiene el elemento y el tipo de datos del nodo. A medida que se va operando se va conociendo el tipo de dato que nos queda. Por ejemplo, en el árbol se ve como la suma entre el int "3" y el float "z" da como resultado un valor de tipo float. Esto se llama síntesis de datos. En el ejemplo, al aplicar la regla 2 y reducir la suma, se ejecuta la llamada a la función que nos indica el tipo de dato del resultado.
En la mayoría de los lenguajes, dado dos tipos de datos tenemos el mismo resultante independientemente del tipo de operación que se está realizando. Estos tienen una tabla de las siguientes características:

	int	float	string
int	int	float	142
float	float	float	143
string	142	143	string

En este ejemplo vemos como cada par de tipo de datos tiene un resultado. En el caso del String el ejemplo supone que operar con cualquier tipo numérico da error y el número en la tabla es el código de error.
Esta tabla puede crecer en compilación cuando el lenguaje tiene sobrecarga de operadores (con nuevos uso para el +, -, *, etc). En caso contrario, de no poderse definir, la tabla es fija. Es decir, esto depende de que si el nuevo tipo de dato pueda operarse o no. Por ejemplo, podemos crear dos tipos de datos nuevos, peras y bananas, pero

si estas no pueden sumarse o restarse no tiene sentido saber qué tipo de dato es el resultante porque nunca lo necesitaremos.
En el árbol puede ser más detallado en cuanto a las conversiones que genera el código. Por ejemplo, este es el árbol más detallado del ejemplo anterior:



Es importante tener clara la diferencia de operar un "int" y un "float" al momento de generar código porque por ejemplo, el primero se suma en CPU y el segundo en el coprocesador matemático. Si no se utiliza este árbol, se deben tener en cuenta las conversiones al momento de generar el Assembler.
Para facilitar esta tarea, los compiladores amplían la tabla que vimos más arriba agregando además del tipo de dato resultante, si se debe convertir alguno de los dos operandos:

	int			float			string		
int	int	-	-	float	C(int to float)	-	142	-	-
float	float	-	C(int to float)	float	-	-	143	-	-
string	142	-	-	143	-	-	string	-	-

Como vemos, cuando el primer elemento es un "Int" y el segundo es un "float", la primer columna de las tres indica el tipo de dato resultante que es "float", la segunda indica que el primer elemento debe hacer una conversión desde "int" a "float" y la tercera columna indica que el segundo operando queda tal cual está.
Puede darse que al operar "long" y "float", como son tipo de datos que como resultado pueden dar algo más grande que no entra en los tipos de datos de los operandos, tengamos algo así:

	float		
long	double	C(long to double)	C(float to double)

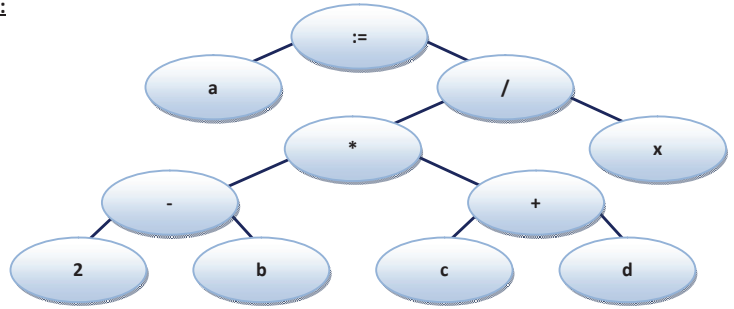
Traducción de Árbol Sintáctico a Assembler

NOTA: En esta sección se considerará que no se realizan conversiones de tipo y todos los elementos son "int".

Vamos a realizar un método que utiliza variables auxiliares y no nos vamos a meter con los diferentes tipos de datos que puede tener el lenguaje. Realizaremos la traducción del siguiente árbol:

```
a := (2 - b) * (c + d) / x
```


Árbol original:



Lo primero que se debe hacer es recorrer el árbol en busca del nodo que tenga los dos hijos lo mas a la izquierda posible. En el ejemplo ese nodo es el de la resta.

Luego se genera el código para ese nodo.

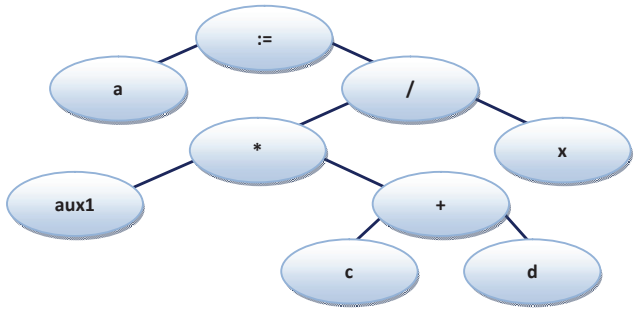
```
MOV R1, 2
SUB R1, b
MOV aux1, R1
```

Tanto "b" como "aux1" son dos posiciones de memoria (estamos suponiendo que es un lenguaje estático). Vale aclarar que hay porciones de esta traducción que son siempre fijas, son las marcadas a continuación:

```
MOV R1, 2
SUB R1, b
MOV aux1, R1
```

Luego de esto se debe dar de alta a "aux1" en la tabla de símbolos. También se elimina el nodo atendido y se reemplaza por uno sin hijos que hace referencia a la variable resultante "aux1".
Volvemos a comenzar el ciclo eligiendo el nodo con hijos más a la izquierda dentro de nuestro árbol resultante:

Árbol Paso 1:

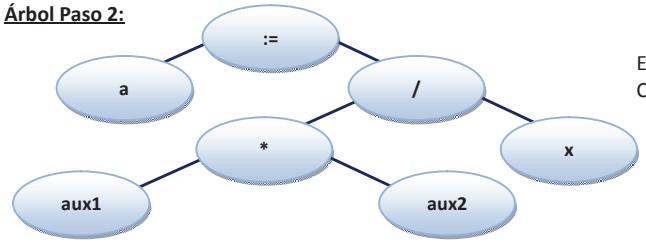


Entonces elegimos el nodo de la suma, el cual genera el siguiente código:

```
MOV R1, c
ADD R1, d
MOV aux2, R1
```

Es un código muy similar al anterior con la diferencia que los operandos y la operación son otros y genera otra variable auxiliar que también es creada en la tabla de símbolos.

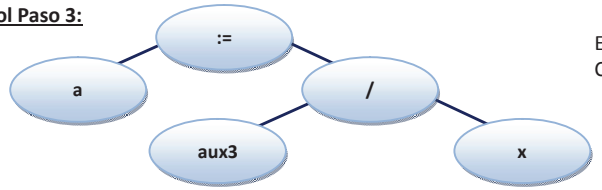
Árbol Paso 2:



El nodo elegido es el de la multiplicación.
Código resultado:

```
MOV R1, aux1
MUL R1, aux2
MOV aux3, R1
```

Árbol Paso 3:



El nodo elegido es el de la división.
Código resultado:

```
MOV R1, aux3
DIV R1, x
MOV aux4, R1
```

Árbol Paso 4:

El nodo elegido es el de la asignación, de hecho es el último. Código resultado:

```
MOV R1, aux4
MOV a, aux4
```

Y finalizamos. Todo este código que fuimos generando es el resultado de la traducción del árbol a Assembler.

Existen compiladores que hacen una post-optimización repasando las instrucciones y eliminando lo que pueda estar demás.

Seguimiento de Registros

Hay compiladores que utilizan registros del CPU en vez de variables auxiliares mientras estos estén libres. A esta técnica se la llama seguimiento de registros porque el compilador debe saber que registros están siendo utilizados con valores intermedios para seguir construyendo las sentencias de operación. Para este fin, se tiene una tabla que indica para cada registro si está libre u ocupado:

Registro	Estado
R1	Libre
R2	Libre
R3	Libre
R4	Libre

Al realizar cada uno de los pasos anteriormente descriptos, el compilador busca el primer registro libre y lo utiliza. Luego marca como ocupado el registro que utilizó y continúa.
Veamos el mismo ejemplo anterior aplicado con esta técnica:

```
a := (2 - b) * (c + d) / x
```


Paso 1: Se genera el código para el nodo de la resta.

```
MOV R1, 2
SUB R1, b
```

Usamos R1 porque es el primer registro marcado como libre en la tabla. →
Luego de esto lo ponemos como ocupado.

Registro	Estado
R1	Ocupado
R2	Libre
R3	Libre
R4	Libre

Paso 2: Se genera el código para el nodo de la suma.

```
MOV R2, c
ADD R2, d
```

Usamos R2 porque es el primer registro marcado como libre en la tabla. →
Luego de esto lo ponemos como ocupado.

Registro	Estado
R1	Ocupado
R2	Ocupado
R3	Libre
R4	Libre

Paso 3: Se genera el código para el nodo de la multiplicación.

```
MUL R1, R2
```

Liberamos el R2 ya que hemos guardado el resultado en el R1. →

Registro	Estado
R1	Ocupado
R2	Libre
R3	Libre
R4	Libre

Paso 4: Se genera el código para el nodo de la división.

```
DIV R1, x
```

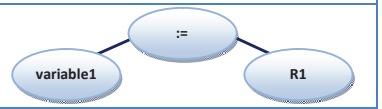
Paso 5: Se genera el código para el nodo de la asignación.

```
MOVa, R1
```

Liberamos el R1 dejándonos todos los registros libres. →

Registro	Estado
R1	Libre
R2	Libre
R3	Libre
R4	Libre

Y finalizamos. Como podemos ver, con esta técnica se ahorra muchísimas instrucciones y variables auxiliares. Pero no evita el uso de variables auxiliares si la sentencia tiene muchos términos (la cantidad de registros es limitada).
Dependiendo de dónde se encuentra las variables que se van a operar el compilador se da cuenta de debe tomar un registro o liberarlo. Veamos los casos:

Explicación	Nodo
Este nodo siempre ocupa un registro nuevo.	
Este nodo siempre libera el R2.	
Este nodo no realiza ninguna ocupación ni liberación.	
En este caso, si la operación es conmutativa no ocurre nada (trabajaría como el ejemplo anterior, invirtiendo los elementos). Pero si no lo es, primero se ocupa un nuevo registro con el valor de la variable, se opera con R1 sobre ese registro nuevo y se libera R1. El valor queda en el nuevo registro.	
Este nodo siempre libera el R1. Al terminar la operaciónde asignación, no queda ningún registro ocupado.	

NOTA:
Al generar Assembler no se debe utilizar el nombre de las variables que le dio el usuario. Veamos un ejemplo:
El usuario creo la siguiente sentencia en nuestro lenguaje:

```
MOV := ADD;
```

Si utilizáramos tal cual los nombres, nuestro compilador generaría lo siguiente:

```
MOV MOV, ADD
```

Esto provocaría un error. Para evadir este problema, debemos agregarle algo más a los nombres de variables elegidos por los usuarios, por ejemplo, el carácter $ adelante, para así no tener problemas:

```
MOV $MOV, $ADD
```

Optimización de Código

El objetivo principal de la optimización de código es hacer que el código final (por ejemplo en Assembler) sea más eficiente.

Reducción Simple

Este método de optimización consiste en realizar todas las operaciones en las que participan constantes lo antes posible, para optimizar la cantidad de instrucciones necesarias en el código final.

Ejemplo: `Z := 3 * 9 * k`
En esta sentencia se puede aplicar la optimización de Reducción Simple, tomando el valor de "3 * 9" y compilandolo siguiente:
Sentencia optimizada: `Z := 27 * k`

Redundancia

Este método consiste en detectar las porciones de código que realizan operaciones repetitivas con el fin de realizar dichas operaciones una única vez y luego utilizar el resultado las veces que sean necesarias.

Ejemplo: `Z := a * b * c + 4 * a * b`
En esta sentencia se puede aplicar la optimización de Redundancia, tomando el valor de "a * b" para evitar que la misma multiplicación se realice dos veces, con una alcanza.

Reordenamiento de Instrucciones

Al reordenar instrucciones, se optimiza el código al reducir el uso de variables auxiliares (registros en el procesador) y la cantidad de instrucciones finalesya que intenta aplicar la nueva operación sobre el resultado que se obtuvo inmediatamente antes.

Ejemplo: `Z := a + b + c * d * (e + f + g * h)`

Sentencia optimizada: `Z := (h * g + f + e) * d * c + b + a`