RESUMEN DE MAK – LENGUAJES Y COMPILADORES (Segundo Cuatrimestre)

Autor: Ariel Nader – arielnader@gmail.com

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).

Compilación Monolítica: Se le llama asi 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.

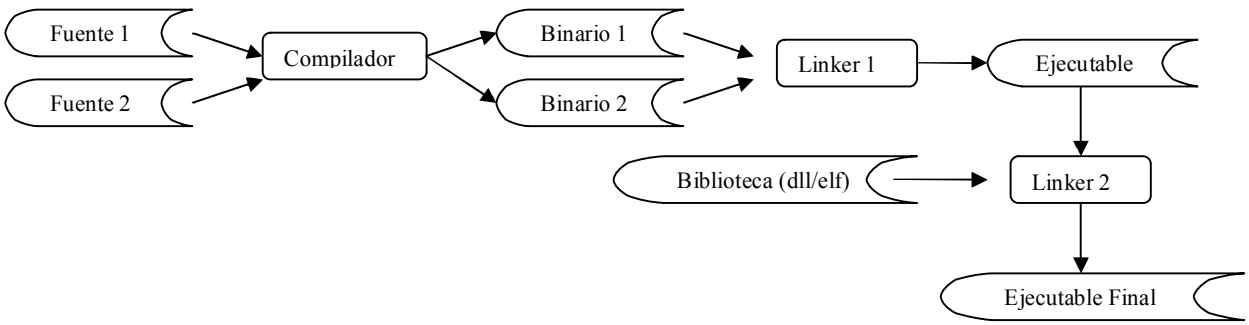
Para esta forma de compilar no se puede descartar por completo la tabla de símbolos luego de la compilación ya que una referencia desde uno de los fuentes puede estar apuntando a otro. Por ejemplo, si tengo un procedimiento "CalcularPrecio" en el fuente 2 y es llamado desde el fuente 1, al momento de compilar este último debo saber donde 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.

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 Compilers: Son compiladores que corren en una máquina y la salida esta preparada para otra máquina. Por ejemplo un compilador de C en Sun para una máquina IBM.

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

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

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

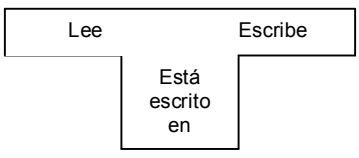
Este experimento no se logró hacer con éxito pero si se pudo generar Analizadores Lexicos y Sintácticos automáticamente. El producto mas conocido que realiza esta tarea es YACC.

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 interpretes de esa máquina que ejecutaban sobre máquinas reales. Esta idea finalmente murió pero tiempo después resurgió con la aparición de Java.

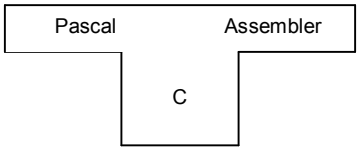
Diagramas T: Para explicar en que consiste un diagrama T primero vamos a hablar de ciertos conceptos. En la construcción de un compilador hay tres lenguajes:

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:

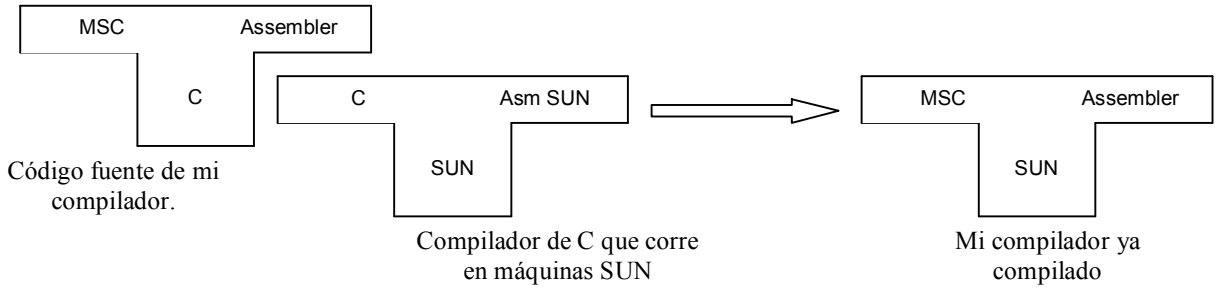


Por ejemplo, un compilador de lenguaje Pascal que está escrito en C y genere una salida en Assembler se representa con el siguiente diagrama T:



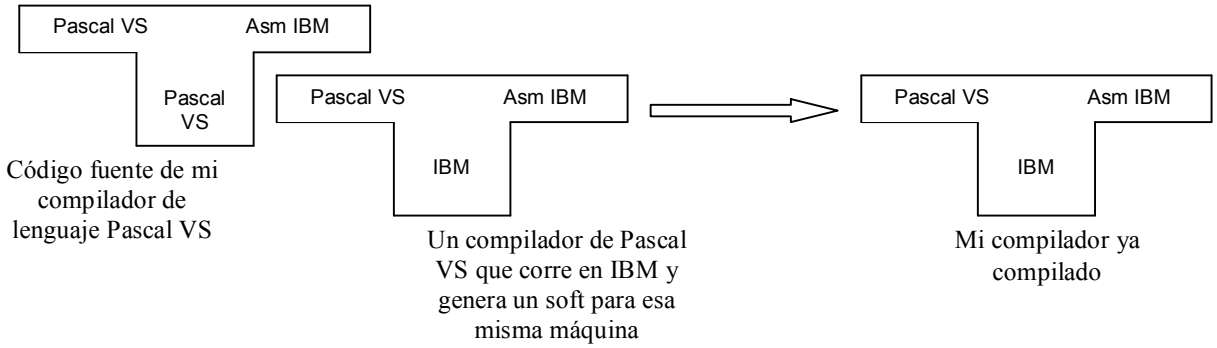
Podemos hacer un diagrama T del compilador que estamos creando y también podemos hacer el diagrama T del compilador con el que estamos compilando nuestro compilador.

Ejemplos (MSC es el nombre de un lenguaje determinado).

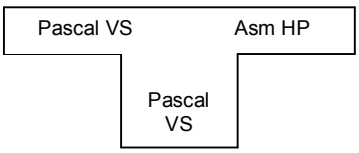


En este esquema se representa la acción de agarrar el código fuente del compilador que estoy creando (primera T) y compilarlo con un compilador tradicional de C que corre en máquinas SUN (segunda T) y obtener como resultado mi compilador pero en binario listo para usar (tercer T). Observe que cuando 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 que lenguaje fue creado.

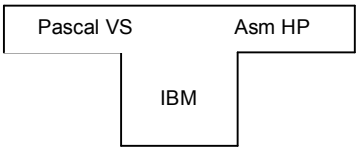
A continuación mostramos un ejemplo con un autocompilador ya que fue escrito en Pascal VS y lee fuentes de Pascal VS:



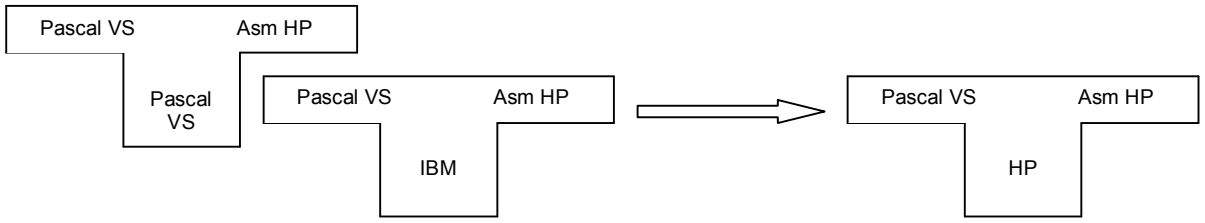
Supongamos que queremos que nuestro compilador corra en máquinas HP y genere código HP. Entonces lo primero que tenemos que hacer es reescribir la salida y que genere código HP, quedando:



Luego como segundo paso es compilar nuestro desarrollo con nuestro compilador de Pascal VS para IBM dándonos como resultado:



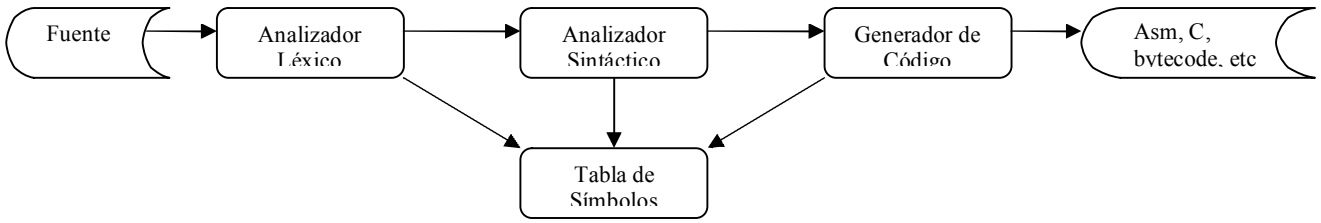
Si ahora compilamos el fuente de nuestro compilador con nuestro compilador en binario que obtuvimos en el paso anterior, nos da como resultado nuestro compilador de Pascal VS corriendo en HP!



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 de un Compilador

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



El analizador léxico divide el programa en fragmentos reconocibles. Por ejemplo:

A := B + 10 * A;

Este seria 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 linea>

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 llaman Tokens. Utiliza una tabla de Tokens donde se enumeran cada no 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.

Luego 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.

Analizador Sintáctico

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.

Parsing Ascendente o Bottom Up

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

- La L significa "Left" porque lee el programa de usuario de izquierda a derecha.
- La R significa "Right" porque para aplicar las reglas busca machear con el lado derecho de la definición en cada regla.

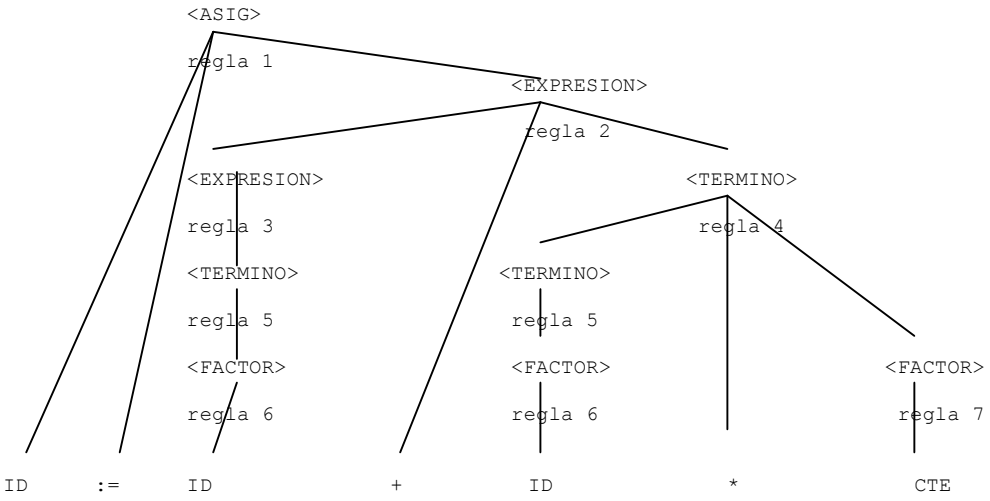
Ejemplo:

Tenemos las siguientes reglas:

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

Y hacemos el árbol de parsing:



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

Parsing Desendente o 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.

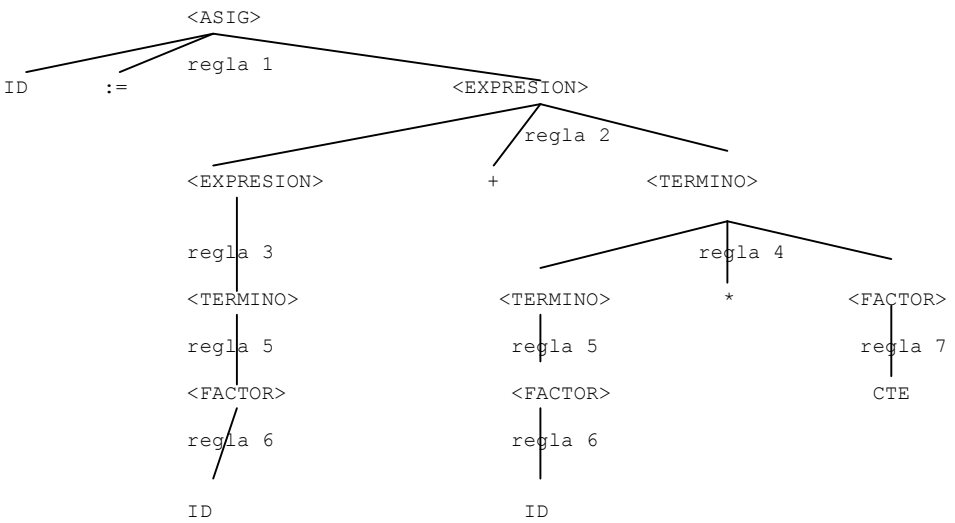
Ejemplo:

Tenemos las siguientes reglas:

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

Y hacemos el árbol de parsing:



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

Método "Recursivo Descendente" (Parsing 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 mas a la izquierda. Ejemplo:

Tenemos las siguientes reglas:

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

```
// Ponemos el símbolo distinguido
<ASIG>

// Usamos regla 1 con el no Terminal <ASIG>
ID := <EXPRESION>

// Aplicamos la regla 2 con <EXPRESION> ya que es el primer no terminal desde la izquierda
ID := <EXPRESION> + <TERMINO>

// Aplicamos la regla 2 con <EXPRESION> nuevamente ya que es el primer no terminal desde la izquierda
ID := <EXPRESION> + <TERMINO> + <TERMINO>

// Aplicamos la regla 2 con <EXPRESION> nuevamente ya que es el primer no terminal desde la izquierda
ID := <EXPRESION> + <TERMINO> + <TERMINO> + <TERMINO>
```

Aca hay un problema! Como el algoritmo lo que hace es aplicar una regla con el no terminal que se encuentre mas a la izquierda, esto entra en un loop 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

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

```
// Ponemos el símbolo distinguido
<ASIG>

// Usamos regla 1 con el no Terminal <ASIG>
ID := <EXPRESION>

// Aplicamos la regla 2 con <EXPRESION>
ID := <TERMINO> + <EXPRESION>

// Aplicamos la regla 4 con <TERMINO>  ya que es el primer no terminal desde la izquierda
ID := <FACTOR> * <TERMINO> + <EXPRESION>

// Luego aplica la regla 6 para tener la primer variable!
ID := ID * <TERMINO> + <EXPRESION>
```

```
// Pero nos encontramos con un problema, el símbolo "*" no se encuentra en nuestro código original asi que tenemos que hacer un "Back Tracking" que es volver atrás sobre las reglas aplicadas para probar otro camino. Asi que deshacemos los efectos de la regla 6 y la 4 que hemos realizado y pasamos a hacer la regla 5.
ID := <FACTOR> + <EXPRESION>

// Ahora si, podemos aplicar la regla 6 para <FACTOR>
ID := ID + <EXPRESION>

// Se va pareciendo al código original, ahora aplicamos la regla 2 con <EXPRESION>
ID := ID + <TERMINO> + <EXPRESION>

// Pero si seguimos con esto nos damos cuenta que el segundo símbolo "+" nos va a provocar un "Back Tracking" asi que tenemos que aplicar la regla 3 en el anterior paso
ID := ID + <TERMINO>

// Aplicamos la regla 4 con <TERMINO>
ID := ID + <FACTOR> * <TERMINO>

// Aplicamos la regla 6 con <FACTOR>
ID := ID + ID * <TERMINO>

// Ya casi está, aplicamos la regla 4 con <TERMINO>
ID := ID + ID * <FACTOR> * <TERMINO>

// Esto nos va a producir otro "Back Tracking" en un paso posterior asi que volvamos y apliquemos la regla 5
ID := ID + ID * <FACTOR>

// 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. 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 → [vacio]

Es decir, una expresión es un termino 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". En estas condiciones el algoritmo tiene el siguiente nombre:

Método "LL(1) o Predictivo Desendente" (Parsing Desendente)

Veamos un ejemplo.

Gramática factorizada:

1. <ASIG> → id := <EXPRESION>
2. <EXPRESION> → <TERMINO> <R>
3. <R> → + <EXPRESION>
4. <R> → [vacio]
5. <TERMINO> → <FACTOR><Q>

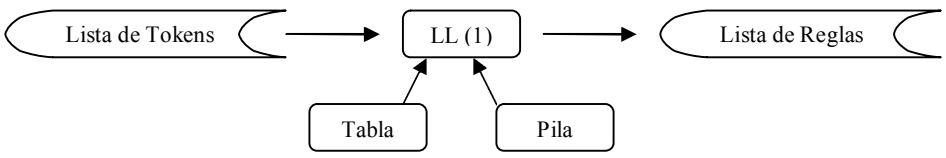
6. <Q> → * <TERMINO>
7. <Q> → [vacio]
8. <FACTOR> → id
9. <FACTOR> → cte

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

```
// Ponemos el símbolo distinguido
<ASIG>

// Usamos regla 1 con el no Terminal <ASIG>
ID := <EXPRESION>

// Aplicamos la regla 2 con <EXPRESION>
ID := <TERMINO> <R>

// Uso regla 5 con <TERMINO>
ID := <FACTOR> <Q> <R>

// Por la regla 8, <FACTOR> es un ID
ID := ID <Q> <R>

// 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>

// 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>

// Usamos regla 2 con <EXPRESION>
ID := ID + <TERMINO> <R>

// Aplicamos la regla 5 con <TERMINO>
ID := ID + <FACTOR> <Q> <R>

// Aplicamos la regla 8 con <FACTOR>
ID := ID + ID <Q> <R>

// Para <Q> usamos la regla 6, ya que tiene el símbolo "*" tal cual se utiliza en el código original
ID := ID + ID * <TERMINO> <R>

// Regla 5 para <TERMINO>
ID := ID + ID * <FACTOR> <Q> <R>

// Regla 9 para <FACTOR>
ID := ID + ID * CTE <Q> <R>

// Regla 7 para <Q>
ID := ID + ID * CTE <R>

// 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 Desendente" (Parsing Desendente)

Vamos a ver la implementación de este método.



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 que 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> → [vacio]
5. <TERMINO> → <FACTOR><Q>
6. <Q> → * <TERMINO>
7. <Q> → [vacio]
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:

Por regla 1 vemos que en la asignación lo primero que aparece es un "id".
Prim(ASIG) = { id }

Como EXPRESION tiene como primer elemento a TERMINO, lo que esté primero en TERMINO será su primero.
Prim(EXPRESION) = Prim(TERMINO) = { id, cte }

Para R, el primero es un signo "+" ya que el [vacío] no se tiene en cuenta.
Prim(R) = { + }

Como TERMINO tiene como primer elemento a FACTOR, lo que esté primero en FACTOR será su primero.
Prim(TERMINO) = Prim(FACTOR) = { id, cte }

Para q, el primero es un signo "*" ya que el [vacío] no se tiene en cuenta.
Prim(Q) = { * }

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:
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) = { $ }

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) = { $ }

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) = { $ }

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) = { +, $ }

Por regla 5 es el siguiente de TERMINO
Sgt(Q) = Sgt(TERMINO) = { +, $ }

Por regla 5, después de FACTOR esta 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 que se explica con la regla 1:

Tomamos la fila de ASIG ya que es el elemento No Terminal que define y luego tomamos la columna del primer elemento Terminal que aparece en la regla después de la flecha de definición → (en este caso "id") y escribimos el contenido de la regla ("Id := <EXPRESION>") .

Para el caso de reglas con "[vacío]" se usan los siguientes del elemento No Terminal que se está definiendo. Ejemplo de esto es la regla 4.

Ejecución del Algoritmo

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

PILA	EXPLICACION	LO QUE FALTA DETECTAR
ASIG	Puse el símbolo distinguido	id := id + id * cte $
EXPRESION := id	Uso la regal 1 metiendola en la Pila (siempre los elementos van al reves)	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 reconocida	$
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 o Bottom 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:

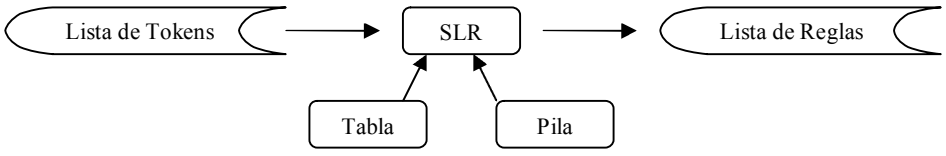
Los métodos son:

- 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.

Implementación del Método "SLR" (Parsing Ascendente)

Vamos a ver la implementación de este método. Aquí tenemos una representación del proceso:



Tenemos la siguiente gramática:

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

Algoritmo de Construcción de la 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 <EXPRESION>:
<EXPRESION> → . <EXPRESION> + <TERMINO>

Ya analicé <EXPRESION>, ahora busco información sobre + <TERMINO>:
<EXPRESION> → <EXPRESION> . + <TERMINO>

Terminé de aplicar la regla:
<EXPRESION> → <EXPRESION> + <TERMINO> .

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:

<ASIG'> → . <ASIG>

Observe que el punto adelante indica que aun no hemos visto el resto de la regla. A continuación debemos definir que elementos válidos son los que pueden llegar a venir a continuación del punto. Como el punto esta delante de un No Terminal, debemos aplicar una regla para resolverlo, a medida que hacemos esto generamos otro estado.

Entonces, teniendo el No Terminal <ASIG>, debemos ver que reglas lo definen:

1. <ASIG> → id := <EXPRESION>

Bueno, sabiendo esto, tenemos que probar que pasa cuando estando en estado 0, nos llega un <ASIG> (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: <ASIG>, "id".

Estado 1 (0, ASIG): Generamos un estado nuevo que surge de recibir el No Terminal <ASIG> estando en el estado 0 (de ahí las dos cosas que están contenidas en el paréntesis). Esto significa que hemos tenido en cuenta el <ASIG> que esperábamos en el estado 0, dejándonos:

<ASIG'> → <ASIG> .

Observemos que el punto avanzo porque ya atendimos la llegada del <ASIG>. 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:

<ASIG> → id . := <EXPRESION>

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.

Elementos válidos a continuación: ":=".

Estado 3 (2, :=): avanzamos por la regla 1, tenemos:

<ASIG> → id := . <EXPRESION>

Bueno, ahora volvemos a tener un No Terminal delante del punto, esta vez es <EXPRESION>. Entonces buscamos las reglas que definen a este No Terminal:

2. <EXPRESION> → . <EXPRESION> + <TERMINO>
3. <EXPRESION> → . <TERMINO>
4. <TERMINO> → . < TERMINO> * <FACTOR>
5. <TERMINO> → . <FACTOR>
6. <FACTOR> → . id
7. <FACTOR> → . cte

Además de las reglas 2 y 3 que definen a <EXPRESION> se incluyeron las reglas 4 y 5 porque una <EXPRESION> es un <TERMINO> según regla 3. Por el mismo motivo se agregaron las reglas 6 y 7 a través de <FACTOR>. Recuerde que <EXPRESION> se incluye por dos motivos en la lista, primero porque es el elemento No Terminal que esta 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: <EXPRESION>, <TERMINO>, <FACTOR>, id, cte.

Estado 4 (3, EXPRESION): Aca hay dos cuestiones, la primera es tomar a <EXPRESION> por haber sido el elemento que en el estado 3 estaba detrás del punto, para lo cual podemos medir el avance:

<ASIG> → id := <EXPRESION> .
Por otro lado, tomamos a <EXPRESION> como el primer elemento de la regla 2 que era la primer posibilidad del estado anterior. Según esto, debemos medir el avance de dicha regla para lo cual estamos asi:

<EXPRESION> → <EXPRESION> . + <TERMINO>

Bueno, ahora debemos definir que elementos vienen a continuación. Si miramos el primer avance, vemos que el punto esta 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, TERMINO): En este caso ocurre lo mismo, tenemos dos avances:

<EXPRESION> → <TERMINO> .

y

<TERMINO> → < TERMINO> . * <FACTOR>

Esto se debe a que en el estado anterior hay dos reglas que comienzan con <TERMINO> 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, FACTOR): En este estado, el avance es:

<TERMINO> → <FACTOR> .

Y acá terminamos con este estado.

Estado 7 (3, id): En este estado, el avance es:

<FACTOR> → id .

Y acá terminamos con este estado.

Estado 8 (3, cte): En este estado, el avance es:

<FACTOR> → cte .

Y acá terminamos con este estado.

Estado 9 (4, +): En este estado, el avance es:

<EXPRESION> → <EXPRESION> + . <TERMINO>

Ahora nos encontramos con el No Terminal <TERMINO>, veamos que reglas puedo aplicar:

4. <TERMINO> → . < TERMINO> * <FACTOR>
5. <TERMINO> → . <FACTOR>
6. <FACTOR> → . id
7. <FACTOR> → . cte

Elementos válidos a continuación: <TERMINO>, <FACTOR>, id, cte.

Estado 10 (5, *): En este estado, el avance es:

<TERMINO> → < TERMINO> * . <FACTOR>

Se viene el FACTOR, veamos que reglas puedo aplicar:

6. <FACTOR> → . id
7. <FACTOR> → . cte

Recordemos que ademas del "id" y la "cte", un elemento que se espera en este estado es <FACTOR> ya que es el No Terminal que esta a continuación del punto.

Elementos válidos a continuación: <FACTOR>, id, cte.

Estado 11 (9, TERMINO): Tenemos:

<EXPRESION> → <EXPRESION> + <TERMINO> .

y

<TERMINO> → <TERMINO> . * <FACTOR>

Como para este No Terminal hay una regla ademas 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, FACTOR)**~~: Este estado finalmente no es tenido en cuenta. Esto es asi porque de aplicar FACTOR al estado 9, el avance quedaria asi:

<TERMINO> → <FACTOR> .

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, FACTOR) va a estar asociado con el estado 6:

Estado 6 (9, FACTOR).

~~**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, FACTOR): El avance es:

<TERMINO> → <TERMINO> * <FACTOR> .

Terminamos con este estado.

~~Estado 13 (10, id)~~: ~~)~~: Pasa lo mismo que en las repeticiones anteriores 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 esta compuesta por tantas filas como estados hayamos generado (en este ejemplo son 13 estados) y en cuanto a las columnas esta dividida en dos, el primer grupo para los No Terminales y el segundo para los Terminales.

	id	:=	+	*	cte	$	ASIG	EXPRESION	TERMINO	FACTOR
0										
1										
2										
3										
4										
5										
6										
7										
8										
9										
10										
11										
12										

La parte sombreada de celeste se le llama “Acción” y la parte gris “Goto”.

Comenzamos a llenar la parte de “Goto”, para ello vamos a detectar aquellos estados en los que se llega cuando recibe un Terminal. Estos son:

Estado 1 (0, ASIG)
Estado 4 (3, EXPRESION)
Estado 5 (3, TERMINO)
Estado 6 (3, FACTOR)
Estado 11 (9, TERMINO)
Estado 6 (9, FACTOR)
Estado 12 (10, FACTOR)

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 ASIG el valor será 1. Completado queda de la siguiente manera:

	id	:=	+	*	cte	$	ASIG	EXPRESION	TERMINO	FACTOR
0							1			
1										
2										
3								4	5	6
4										
5										
6										
7										
8										
9									11	6
10										12
11										
12										

Esta sección que se llama "Goto" básicamente lo que nos brinda es la posibilidad de saber a que estado debemos ir cuando el algoritmo del analizador sintáctico SLR debe recurri reconoce un No Terminal y 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:

<ASIG> → id := <EXPRESION> .

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

A continuación vamos a comenzar a llenar la sección de "Acciones". Para ello 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 <EXPRESION> + <TERMINO> como una <EXPRESION>.

Comenzaremos a llenar los casilleros para los desplazamientos. Para esto 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 debe ubicar 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". Completado queda de la siguiente manera:

	id	:=	+	*	cte	$	ASIG	EXPRESION	TERMINO	FACTOR
0	D2						1			
1										
2		D3								
3	D7				D8			4	5	6
4			D9							
5				D10						
6										
7										
8										
9	D7				D8				11	6
10	D7				D8					12
11				D10						
12										

A continuación vamos a completar las celdas para las reducciones. Para eso 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, ASIG): <ASIG'> → <ASIG> .
Estado 4 (3, EXPRESION): <ASIG> → id := <EXPRESION> .
Estado 5 (3, TERMINO): <EXPRESION> → <TERMINO> .
Estado 6 (3, FACTOR): <TERMINO> → <FACTOR> .
Estado 7 (3, id): <FACTOR> → id .
Estado 8 (3, cte): <FACTOR> → cte .
Estado 11 (9, TERMINO): <EXPRESION> → <EXPRESION> + <TERMINO> .
Estado 6 (9, FACTOR): <TERMINO> → <FACTOR> .
Estado 7 (9, id): <FACTOR> → id .
Estado 8 (9, cte): <FACTOR> → cte .
Estado 12 (10, FACTOR): <TERMINO> → <TERMINO> * <FACTOR> .
Estado 7 (10, id): <FACTOR> → id .
Estado 8 (10, cte): <FACTOR> → cte .

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

Sgt(ASIG') = { $ }
Sgt(ASIG) = Sgt(ASIG') = { $ }
Sgt(EXPRESION) = Sgt(ASIG) U { + } = { $, + }
Sgt(TERMINO) = Sgt(EXPRESION) U { * } = { $, +, * }
Sgt(FACTOR) = Sgt(TERMINO) = { $, +, * }

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.

La tabla quedaría asi:

	id	:=	+	*	cte	$	ASIG	EXPRESION	TERMINO	FACTOR
0	D2						1			
1						R0				
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				

Por último vamos a hacer un retoque. Es lógico pensar que si aplicamos la regla 0 en algun momento del algoritmo podemos decir que el programa esta 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:

	id	:=	+	*	cte	$	ASIG	EXPRESION	TERMINO	FACTOR
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 un símbolo que será nuestro elemento activo y en la última columna tendremos nuestra entrada que iremos consumiendo hasta finalizar.

PILA	SIMBOLO	EXPLICACION	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] y encontramos D2. Con lo cual consumimos el id y quedamos en estado 2.	:= id + id * cte $
0 2 3	id :=	Recibimos el :=, con [2, :=] = D3, desplazamos y 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 el hay un reducir porque [7, +] = R6. La regla 6 indica F → id, con lo cual tomo el último id y lo transformo a F. Además debo quitar un elemento de estado de la pila, quedándome con el estado anterior que es 3. Luego, utilizo la tabla de GOTO con el estado actual y el elemento No Terminal con el que se ha reducido, esto da [3, F] = 6, y este último es el nuevo estado. Observe que el símbolo + no ha sido consumido porque no hubo desplazamiento.	+ id * cte $
0 2 3 5	id := T	Vuelvo a recibir el +, tenemos [6, +] = R5. Reducimos por regla 5, que es T → F, quitamos el estado 6 y hacemos [3, T] = 5, este último es 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, quitamos el estado 5 y hacemos [3, E] = 4, este último es 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, consumimos el *.	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 quitar estados, no vamos a quitar uno solo sino los últimos 3 estados ya que la definición de la regla tiene 3 elementos (T * F), con lo cual quedamos en el estado 9. Hacemos [9, T] = 11 y ese 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 3 estados quedándonos con el estado 3, y hacemos [3, E] = 4.	$
0 1	A	Recibimos $, con [4, $] = R1, usamos regla 1 que es A → id := E, reducimos y quitamos los 3 estados quedándonos con el 0. Con [0, A] = 1.	$
		Recibimos $, con [1, $] = OK, con lo cual aplicamos regla 0 (A' → A) Y finalizamos concluyendo que la sentencia no tiene errores sintácticos.	$

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

Generación de Código

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.

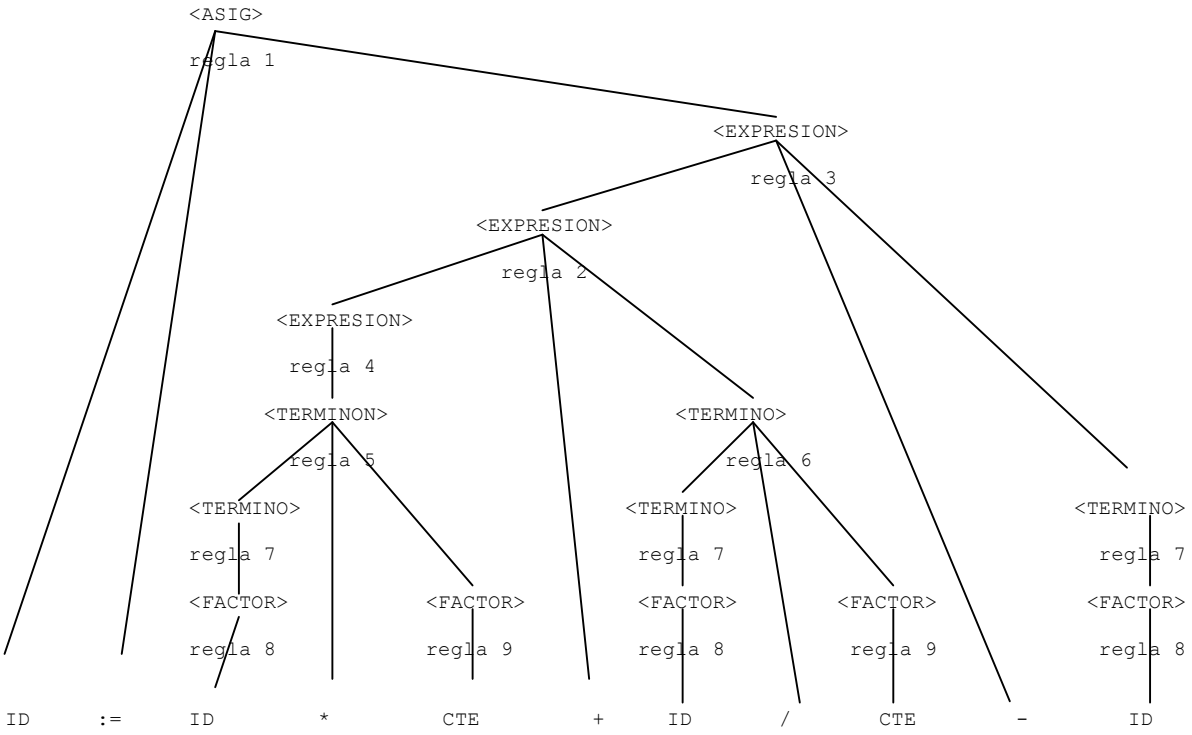
Vamos a usar esta gramática de ejemplo:

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

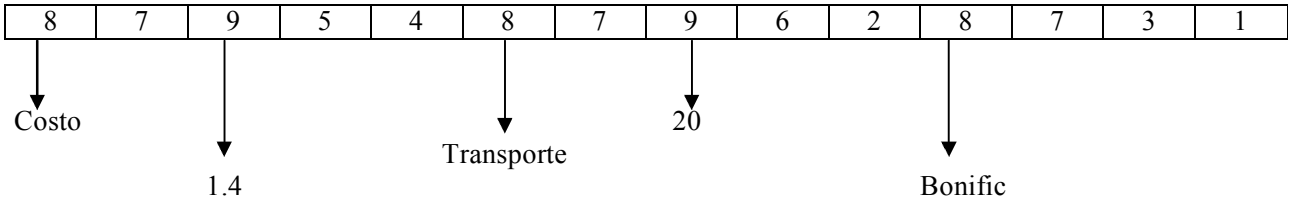
La sentencia puede verse como:
id := id * cte + id / cte – id

Si hacemos el árbol de parsing:



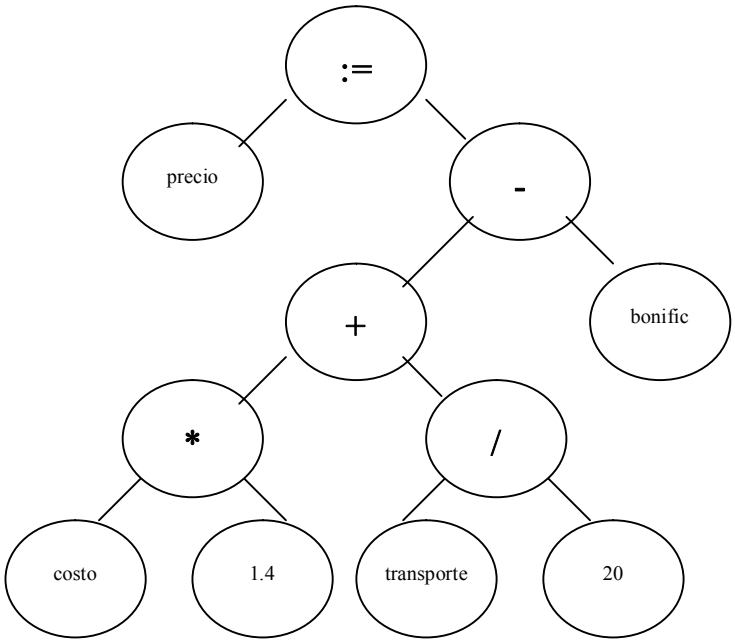
Tira de reglas aplicadas: 8, 7, 9, 5, 4, 8, 7, 9, 6, 2, 8, 7, 3, 1.

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:



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

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 asi 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.

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".

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 intermendias:

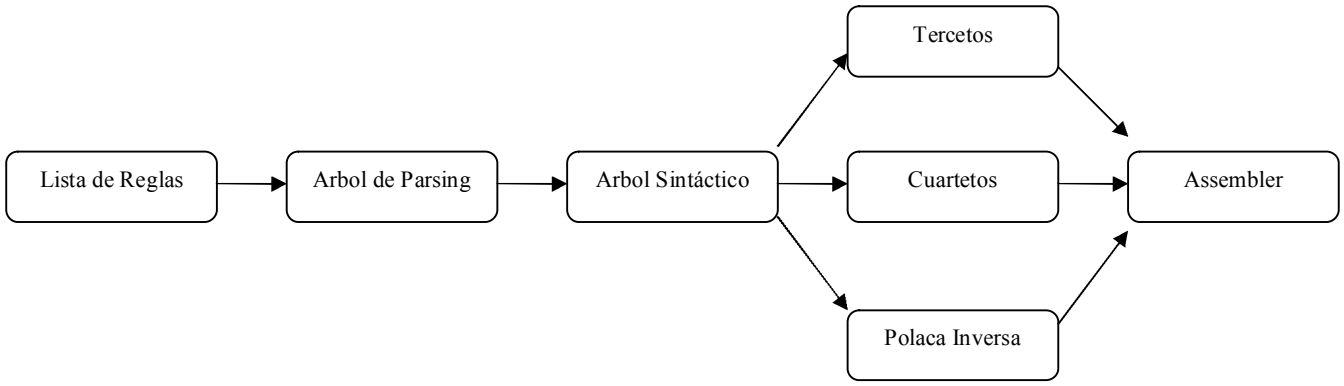
1) (*, costo, 1.4)
2) (/, transporte, 20)
3) (+, "1", "2")
4) (-, "3", bonific)
5) (:=, precio, "4")

Los valores entre comillas son referencias a otro de los tercetos.

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, …)

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 Arbol 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 al objetivos que tengamos, los cual pueden ser:

- Se quiere obtener un compilador rápido y compacto;
  Etapas:
  1. Lista de Reglas
  2. Assembler

- Se quiere que nuestro compilador obtenga ejecutables eficientes;
  Etapas:
  1. Lista de Reglas
  2. Arbol Sintáctico
  3. Tercetos
  4. Assembler

- Poco esfuerzo a la hora de desarrollar nuestro compilador;
  Etapas:
  1. Lista de Reglas
  2. Polaca Inversa
  3. Assembler

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 mas eficientes.

A continuación vemos dos ejemplos de optimización.

Ejemplo 1:

Z := 3 * 9 * k

En esta sentencia se puede aplicar la optimización de **Reducción Simple**, tomando el valor de "3 * 9" y compilando lo siguiente:

Z := 27 * k

Ejemplo 2:

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.

Traducción de Lista de Reglas a Árbol Sintáctico

A continuación vamos a explicar la forma de construir el árbol sintáctico en memoria. Utilizaremos la gramática de siembre:

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

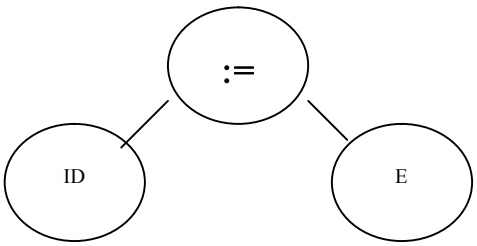
La idea es generar en memoria una serie de nodos que están interconectados. Para eso, cada uno tiene dos punteros, uno al hijo izquierdo y el otro al hijo derecho. A su vez vamos a tener tantos punteros a nodo como No Terminales exista en nuestra gramática. Estos punteros van a ir siendo desplazados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

- Aptr
- Eptr
- Tptr
- Fptr

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:

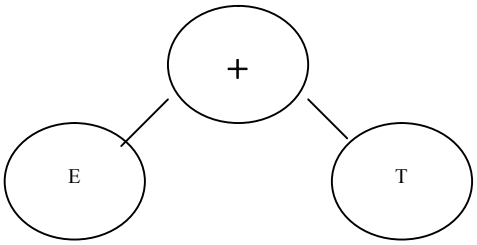
1. <ASIG> → id := <EXPRESION>
   Aptr = crearNodo(":=", crearHoja(id), Eptr)

   Se crea el nodo que contiene al signo de asignación y toma como hijo izquierdo una hoja creada con el "id" reconocido y como hijo derecho la expresión reconocida en Eptr. El nodo resultante queda apuntado por el puntero Aptr. Es decir, construye lo siguiente:



2. <EXPRESION> → <EXPRESION> + <TERMINO>
   Eptr = crearNodo("+", Eptr, Tptr)

   Crea un nuevo nodo para la suma y pone como hijos a la expresión y al termino reconocidos.



3. <EXPRESION> → <EXPRESION> - <TERMINO>
   Eptr = crearNodo("-", Eptr, Tptr)

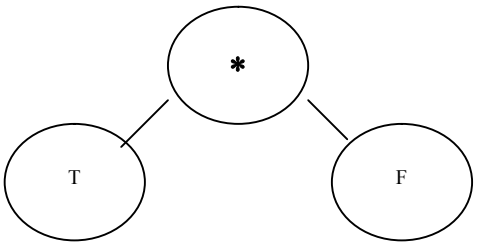
   Parecido al anterior pero con el nodo de resta.

4. <EXPRESION> → <TERMINO>
   Eptr = Tptr

   Con esta sentencia, el nodo que estaba apuntando Tptr (por lo tanto, que hasta ese momento era un Termino) ahora pasa a ser apuntado por Eptr (es decir, ahora pasa a ser una Expresión).

5. <TERMINO> → < TERMINO> * <FACTOR>
   Tptr = crearNodo("*", Tptr, Fptr)

   Se crea el nodo que contiene a la multiplicación y toma como hijo izquierdo al nodo apuntado en Tptr y como hijo derecho al nodo apuntado por Fptr. El nodo resultante queda apuntado por el puntero Tptr. Es decir, construye lo siguiente:



6. <TERMINO> → < TERMINO> / <FACTOR>
   Tptr = crearNodo("/", Tptr, Fptr)

   Parecido al anterior pero con el nodo de división.

7. <TERMINO> → <FACTOR>
   Tptr = Fptr

   Con esta sentencia, el nodo que estaba apuntando Fptr (por lo tanto, que hasta ese momento era un Factor) ahora pasa a ser apuntado por Tptr (es decir, ahora pasa a ser un Termino).

8. <FACTOR> → id
   Fptr = 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. <FACTOR> → cte
   Fptr = crearHoja(cte)

Esta función crea un nuevo nodo en memoria y le pone el valor pasado como parámetro al contenido. En este caso “cte”.

A continuación vamos a comenzar con el seguimiento al algoritmo. Recordemos que estamos compilando la siguiente asignación:

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

Debemos recorrer la lista de reglas resultantes de la sección anterior e ir ejecutando los pasos:

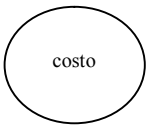
Regla 8: <FACTOR> → id

Se reconoce como un Factor al identificador “precio”. Esto ejecuta la sentencia:

Fptr = crearHoja(“costo”)

Nos queda:

- Aptr: null
- Eptr: null
- Tptr: null
- Fptr: nodo “costo”



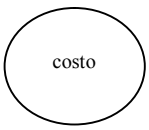
Regla 7: <TERMINO> → <FACTOR>

Se reconoce al Factor que reconocimos último como un Termino. Ejecuta:

Tptr = Fptr

Nos queda:

- Aptr: null
- Eptr: null
- Tptr: nodo “costo”
- Fptr: nodo “costo”



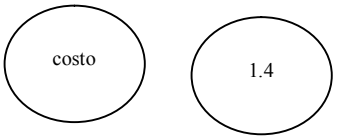
Regla 9: <FACTOR> → cte

Se reconoce como un Factor a la constante “1.4”. Esto ejecuta la sentencia:

Fptr = crearHoja(“1.4”)

Nos queda:

- Aptr: null
- Eptr: null
- Tptr: nodo “costo”
- Fptr: nodo “1.4”



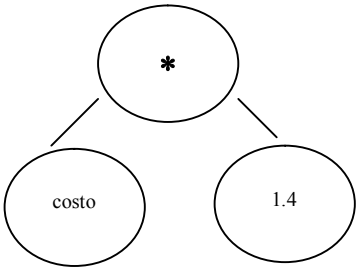
Regla 5: <TERMINO> → < TERMINO> * <FACTOR>

Reduce el Termino y el Factor que habiamos reconocido último generando el nodo de multiplicación. Esto ejecuta la sentencia:

Tptr = crearNodo("*", Tptr, Fptr)

Nos queda:

- Aptr: null
- Eptr: null
- Tptr: nodo "*"
- Fptr: nodo "1.4"



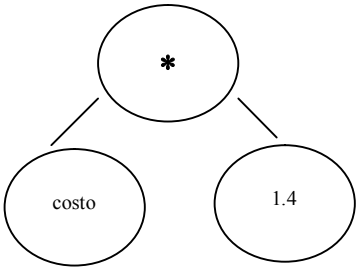
Regla 4: <EXPRESION> → <TERMINO>

Ejecuta:

Eptr = Tptr

Nos queda:

- Aptr: null
- Eptr: nodo "*"
- Tptr: nodo "*"
- Fptr: nodo "1.4"



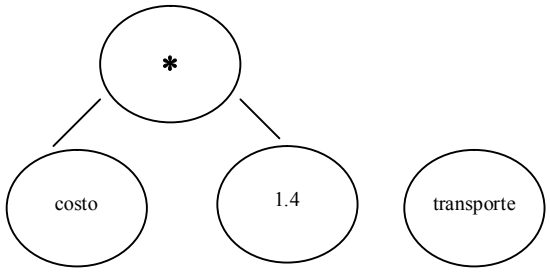
Regla 8: <FACTOR> → id

Ejecuta:

Fptr = crearHoja("transporte")

Nos queda:

- Aptr: null
- Eptr: nodo "*"
- Tptr: nodo "*"
- Fptr: nodo "transporte"



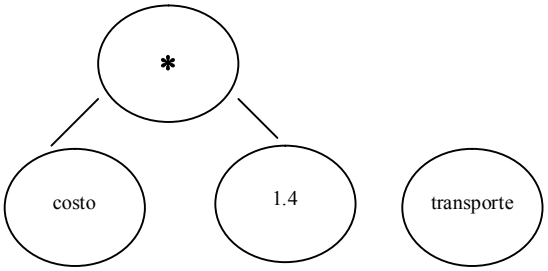
Regla 7: <TERMINO> → <FACTOR>

Ejecuta:

Tptr = Fptr

Nos queda:

- Aptr: null
- Eptr: nodo "*"
- Tptr: nodo "transporte"
- Fptr: nodo "transporte"



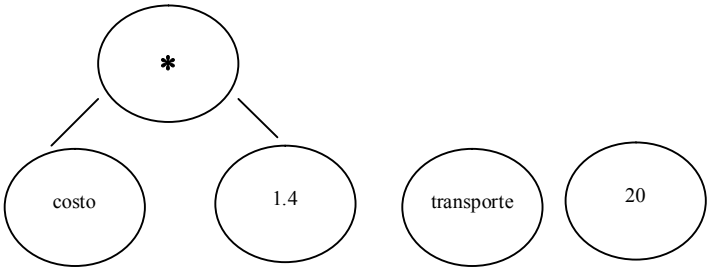
Regla 9: <FACTOR> → cte

Ejecuta:

Fptr = crearHoja("20")

Nos queda:

- Aptr: null
- Eptr: nodo "*"
- Tptr: nodo "transporte"
- Fptr: nodo "20"



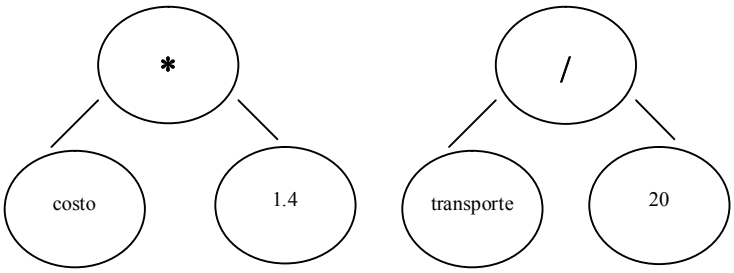
Regla 6: <TERMINO> → < TERMINO> / <FACTOR>

Ejecuta:

Tptr = crearNodo("/", Tptr, Fptr)

Nos queda:

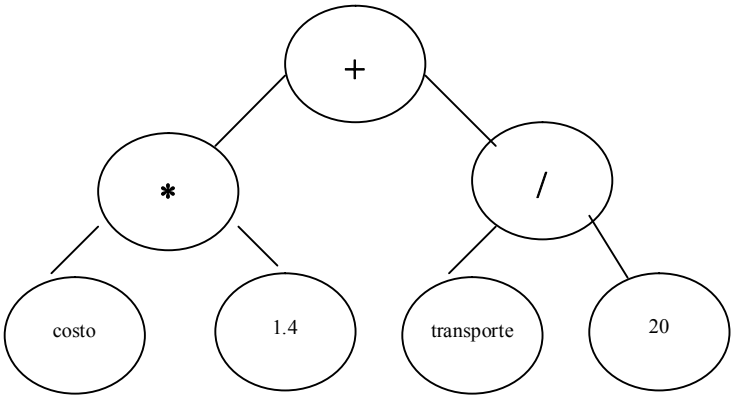
- Aptr: null
- Eptr: nodo "*"
- Tptr: nodo "/"
- Fptr: nodo "20"



Regla 2: <EXPRESION> → <EXPRESION> + <TERMINO>

Ejecuta:

Eptr = crearNodo("+", Eptr, Tptr)



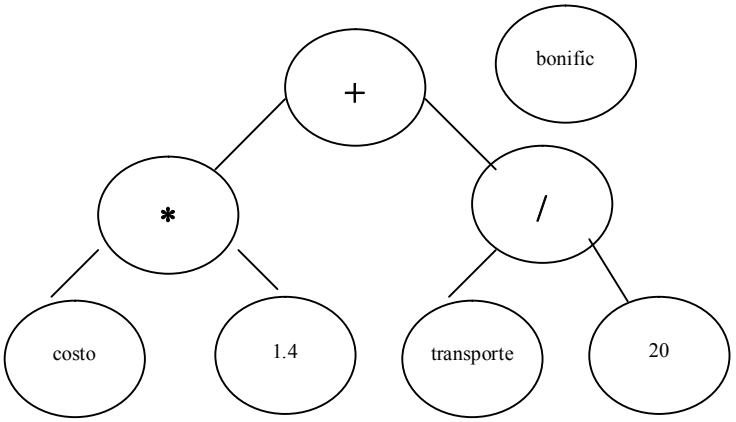
Nos queda:

- Aptr: null
- Eptr: nodo "+"
- Tptr: nodo "/"
- Fptr: nodo "20"

Regla 8: <FACTOR> → id

Ejecuta:

Fptr = crearHoja("bonific")



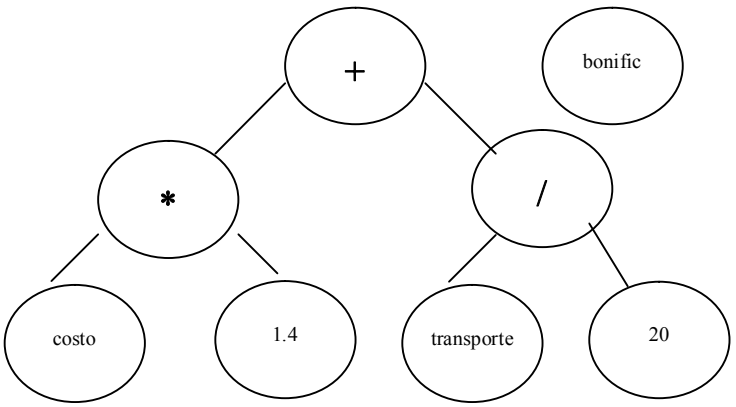
Nos queda:

- Aptr: null
- Eptr: nodo "+"
- Tptr: nodo "/"
- Fptr: nodo "bonific"

Regla 7: <TERMINO> → <FACTOR>

Ejecuta:

Tptr = Fptr



Nos queda:

- Aptr: null
- Eptr: nodo "+"
- Tptr: nodo "bonific"
- Fptr: nodo "bonific"

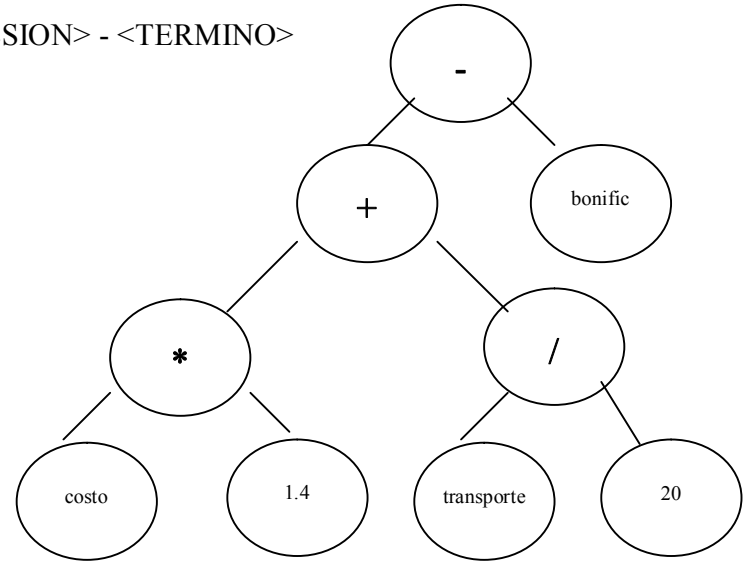
Regla 3: <EXPRESION> → <EXPRESION> - <TERMINO>

Ejecuta:

Eptr = crearNodo("-", Eptr, Tptr)

Nos queda:

- Aptr: null
- Eptr: nodo "-"
- Tptr: nodo "bonific"
- Fptr: nodo "bonific"



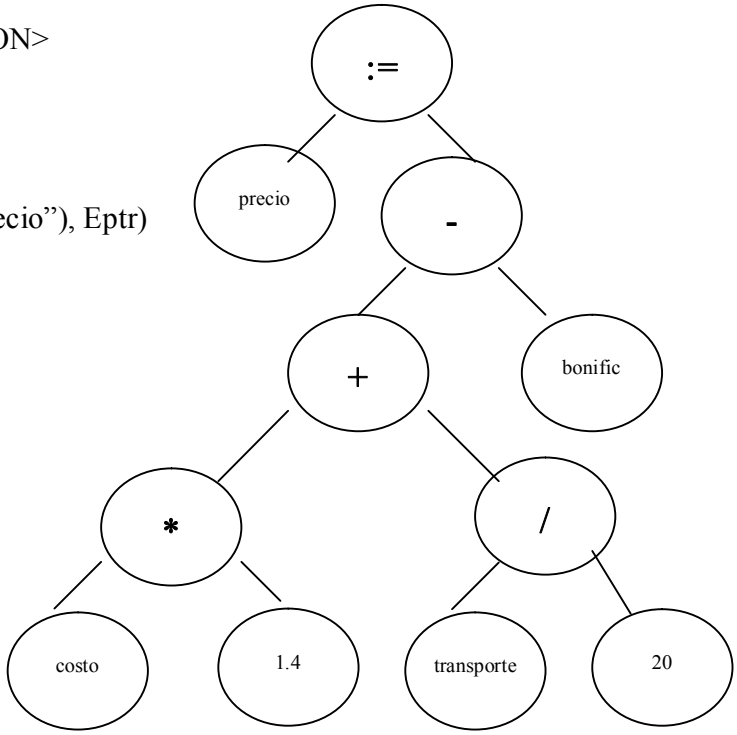
Regla 1: <ASIG> → id := <EXPRESION>

Ejecuta:

Aptr = crearNodo(":=", crearHoja("precio"), Eptr)

Nos queda:

- Aptr: hoha ":="
- Eptr: nodo "-"
- Tptr: nodo "bonific"
- Fptr: nodo "bonific"

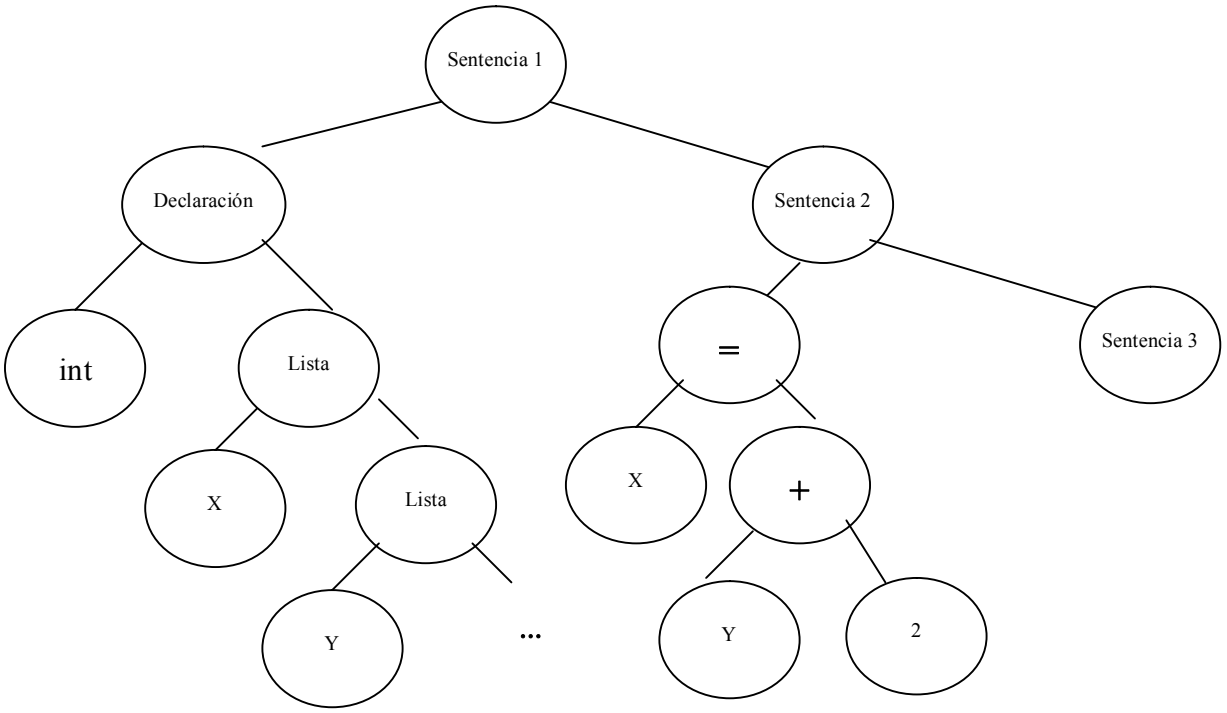


Se acabaron las reglas, por lo tanto termina el algoritmo y el puntero Aptr queda apuntando al comienzo del árbol sintáctico que quedó listo en memoria.

Nota: Tomemos como ejemplo el siguiente código:

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

Sabemos que la primer sentencia, que es de declaración, no va a generar código assembler. Por lo tanto nos podemos preguntar si es necesario armar un árbol sintáctico como el siguiente:



La respuesta es que no esta mal hacerlo pero no es necesario. Lo mas práctico es separar las sentencias ejecutables de las de declaración y dejar que estas últimas se encarguen de generar Tabla de Símbolos.

Tipos de Datos en la 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 que 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 esta compuesta por el puntero y por el tipo de dato.

Entonces serian:

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

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

1. <ASIG> → id := <EXPRESION>
   A.ptr = crearNodo(":=", crearHoja(id, id.tipo), Eptr, id.tipo)

2. <EXPRESION> → <EXPRESION> + <TERMINO>

E.tipo = fun(E.tipo, T.tipo)
E.ptr = crearNodo("+", E.ptr, T.ptr, E.tipo)

3. <EXPRESION> → <EXPRESION> - <TERMINO>
   E.tipo = fun(E.tipo, T.tipo)
   E.ptr = crearNodo("-", E.ptr, T.ptr, E.tipo)

4. <EXPRESION> → <TERMINO>
   E.ptr = T.ptr
   E.tipo = T.tipo

5. <TERMINO> → < TERMINO> * <FACTOR>
   T.tipo = fun(T.tipo, F.tipo)
   T.ptr = crearNodo("*", T.ptr, F.ptr, T.tipo)

6. <TERMINO> → < TERMINO> / <FACTOR>
   T.tipo = fun(T.tipo, F.tipo)
   T.ptr = crearNodo("/", T.ptr, F.ptr, T.tipo)

7. <TERMINO> → <FACTOR>
   T.ptr = F.ptr
   T.tipo = F.tipo

8. <FACTOR> → id
   F.ptr = crearHoja(id, id.tipo)
   F.tipo = id.tipo

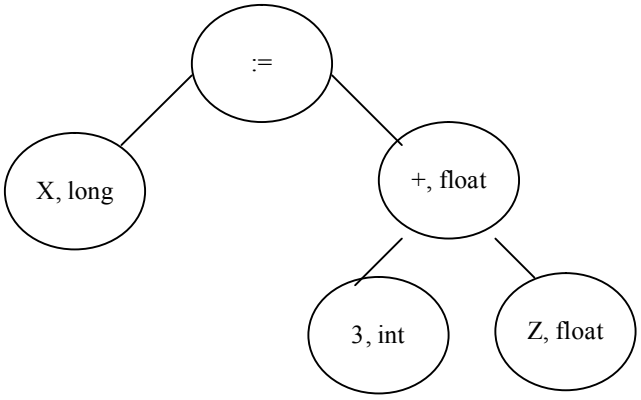
9. <FACTOR> → cte
   F.ptr = crearHoja(cte, cte.tipo)
   F.tipo = cte.tipo

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. Tenemos:

x := 3 + z

El árbol sería el siguiente:



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 vé como la suma entre el entero "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.

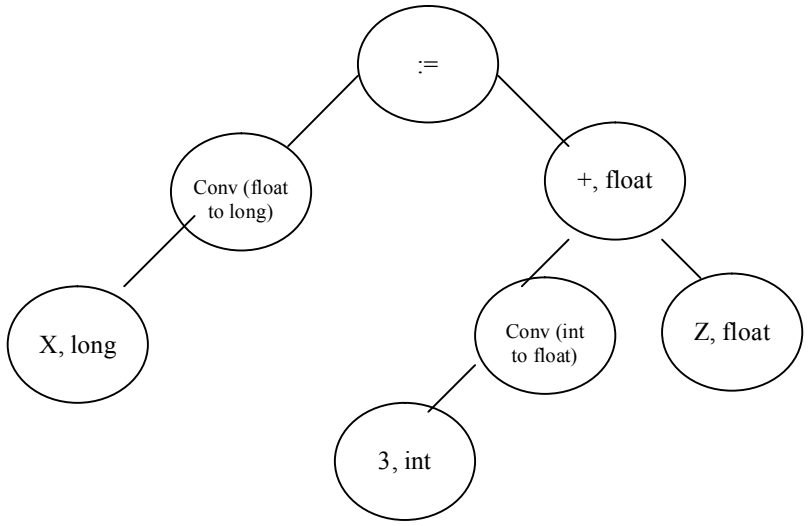
En la mayoría de los lenguajes, dado dos tipo de datos tenemos el mismo resultante independientemente del tipo de operación que se está realizando. Estos tienen una tabla de las siguientes características:

	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 que tipo de dato es el resultante porque nunca lo necesitaremos.

En el árbol puede ser mas detallado en cuanto a las conversiones que genera el código. Por ejemplo, este es el árbol mas 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 amplian la tabla que vimos mas arriba agregando ademas 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 tercer columna indica que el segundo operando queda tal cual está.

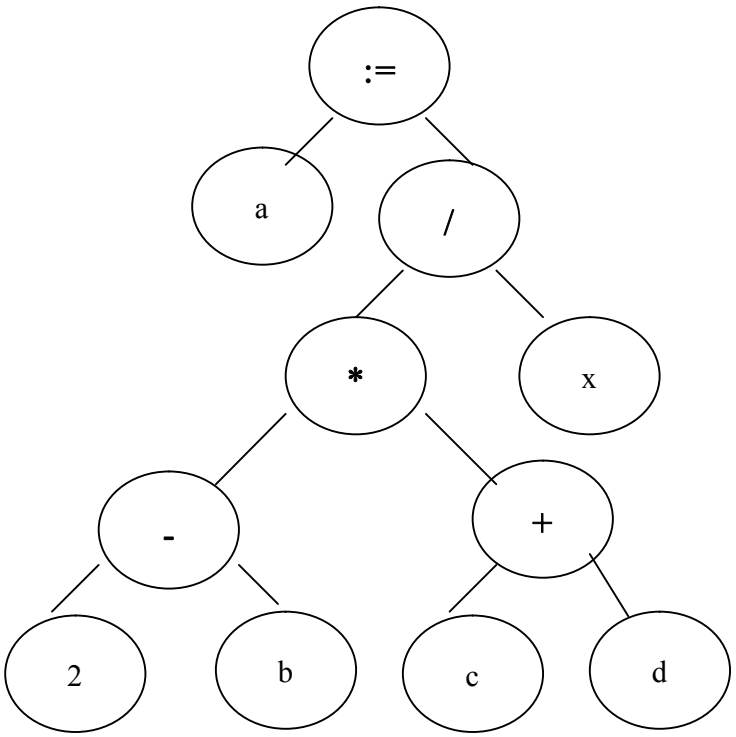
Puede darse que al operar “long” y “float”, como son tipo de datos que como resultado pueden dar algo mas grande que no entra en los tipos de datos de los operandos, tengamos algo asi:

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

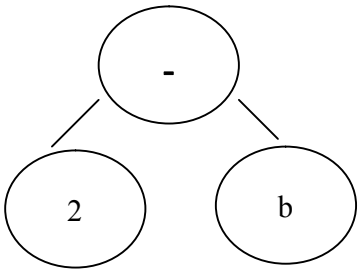
Traducción de Árbol Sintáctico a Assembler (sin conversión de tipos y siendo todos “int”)

Vamos a realizar un método que utiliza variables auxiliares y por ahora 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



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.

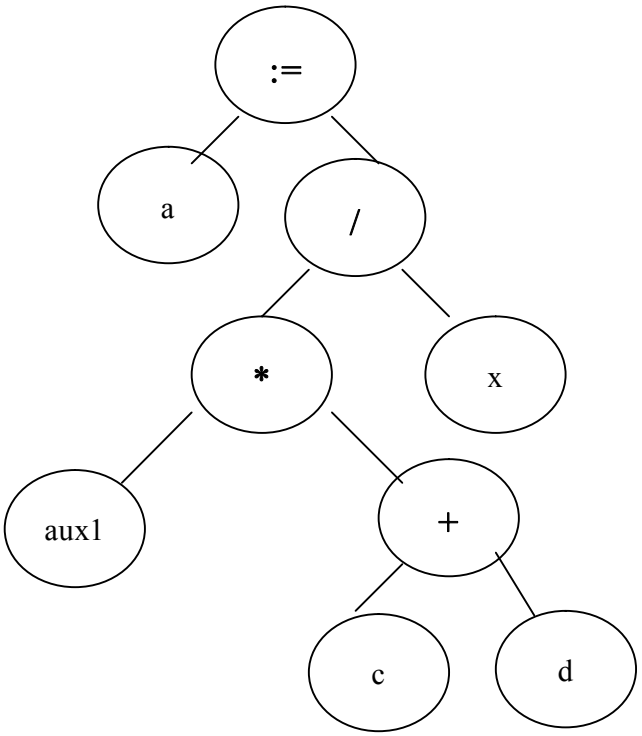
```
MOV    R1, 2
SUB    R1, b
MOV    aux1, R1
```

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:

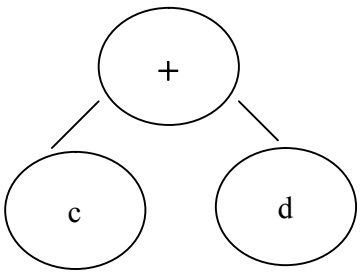
```
MOV    R1, 2
SUB    R1, b
MOV    aux1, R1
```

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 mas a la izquierda dentro de nuestro árbol resultante.



Elegimos el nodo de la suma:

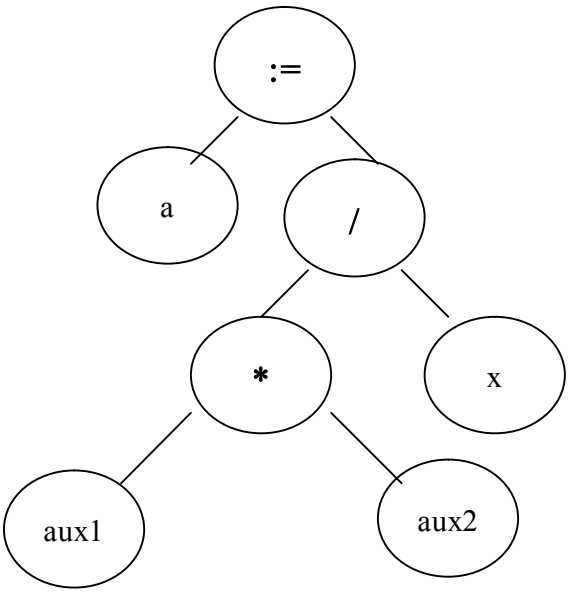


Esto genera el siguiente código:

```
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.

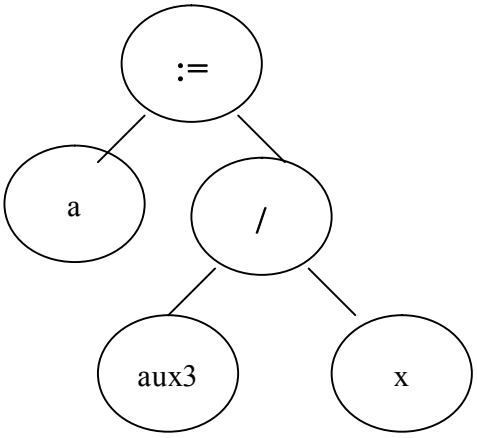
Continuo con lo que falta:



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

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

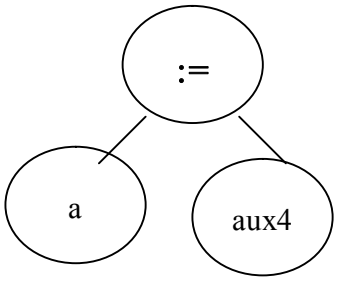
Continuo con lo que falta:



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

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

Continuo con lo que falta:



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 postoptimización repasando las instrucciones y eliminando lo que pueda estar demás.

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 esta 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 continua. Veamos el mismo ejemplo anterior aplicado con esta técnica.

a := (2 – b) * (c + d) / x

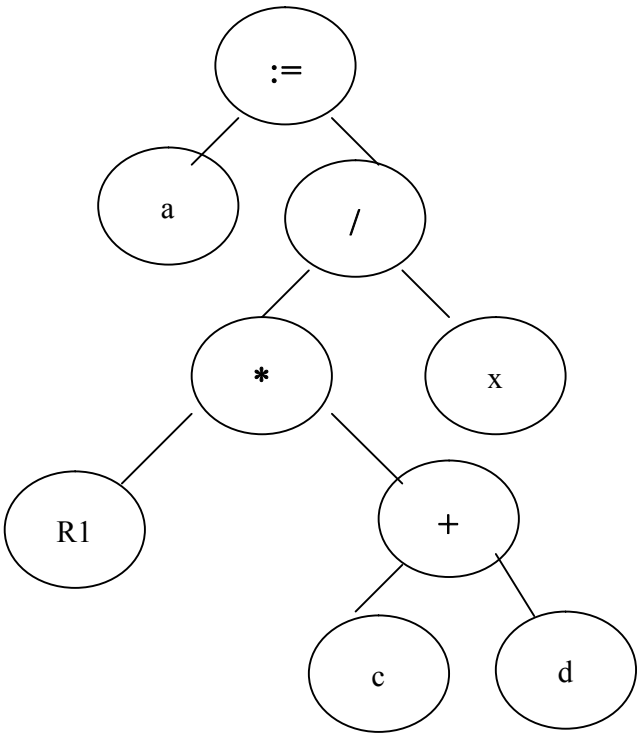
En el primer paso se genera el código para el nodo de la resta.

```
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

Nos queda el árbol de la siguiente manera:



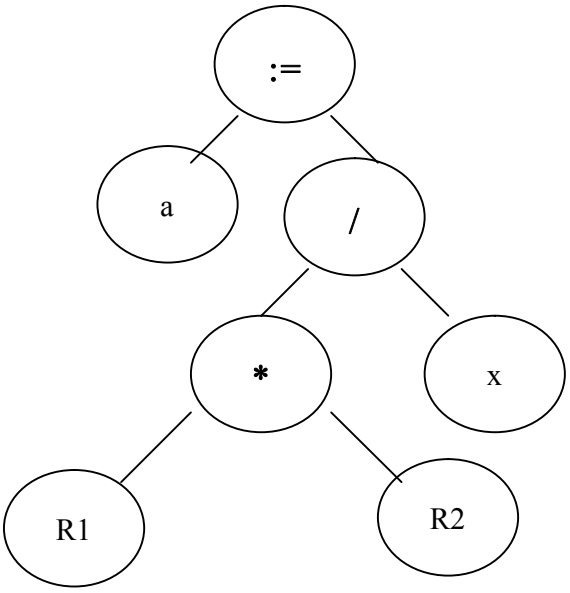
Elegimos el nodo de la suma. Esto genera el siguiente código:

```
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

Nos queda el árbol de la siguiente manera:



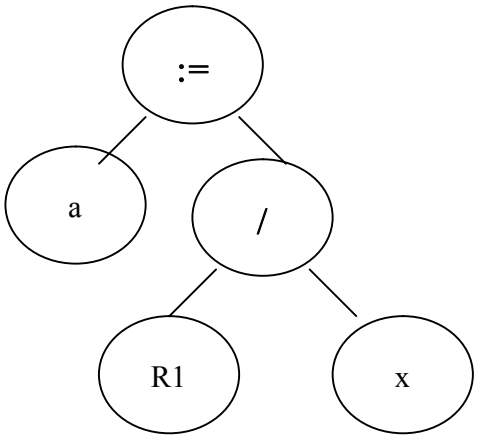
El nodo elegido es el de la multiplicación. Código resultado:

```
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

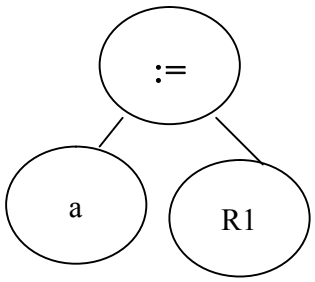
Nos queda el árbol de la siguiente manera:



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

```
DIV    R1, x
```

Continuo con lo que falta:



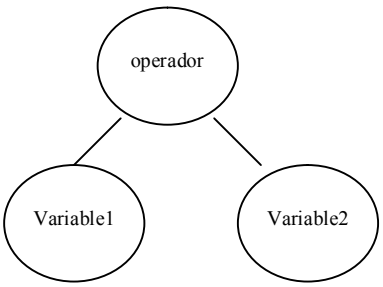
El nodo elegido es el de la asignación, de hecho es el último. Código resultado:

```
MOV    a, R1
```

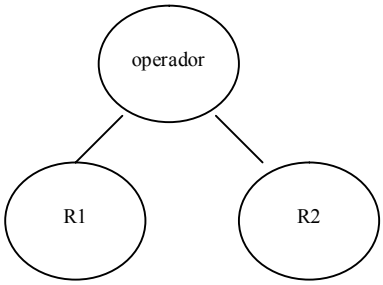
Liberamos el R1 dejándonos todos los registros libres.

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).

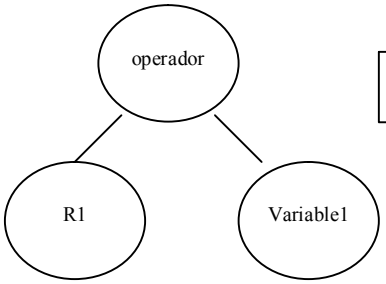
Dependiendo de donde se encuentra las variables que se van a operar el compilador se da cuenta de debe tomar un registro o liberarlo. Veamos los casos:



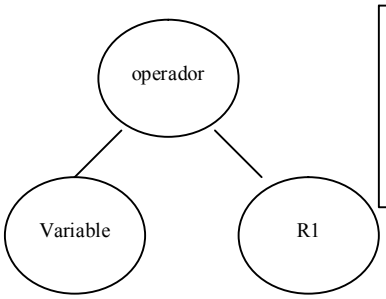
Este nodo siempre ocupa un registro nuevo.



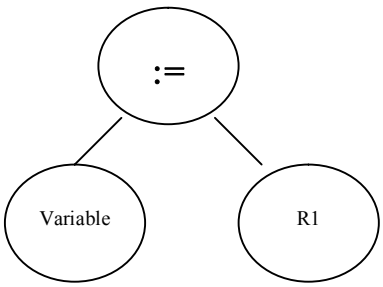
Este nodo siempre libera el R2



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 operación de 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 mas a los nombres de variables elegidos por los usuarios, por ejemplo, el carácter $ adelante:

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

Ahora si, no tenemos problemas.