COMPILADORES E INTÉRPRETES

2019/20

CONTENIDO

1. INTRODUCCIÓN A LA COMPILACIÓN

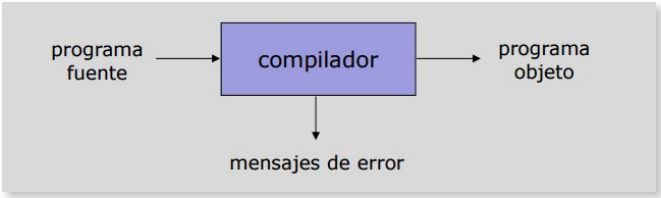
1.1. LENGUAJES DE PROGRAMACIÓN

Hay diversos tipos de lenguajes de programación:

1. **Lenguaje máquina:** Específico de cada máquina. Consiste en secuencias de 0 y 1.
2. **Lenguaje ensamblador:** Específico de cada máquina. Proporciona un conjunto de nombres para instrucciones sencillas de la máquina o posiciones de memoria.
3. **Lenguajes de alto nivel:** Independientes de la máquina. Incorporan estructuras de control, variables con tipo, anidamiento, recursividad y procedimientos.
4. **Lenguajes orientados a problemas:** Específicos de colecciones de problemas, reducen el tiempo de programación, mantenimiento y depuración (SQL).

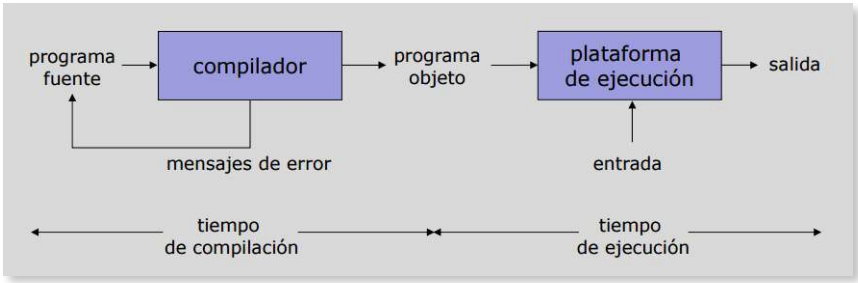
1.2. PROCESADORES DE LENGUAJES

Un **compilador** es un programa que traduce otro programa, escrito en un **lenguaje fuente** a otro programa equivalente escrito en un **lenguaje objeto**. Informa al usuario de errores presentes en el código fuente.

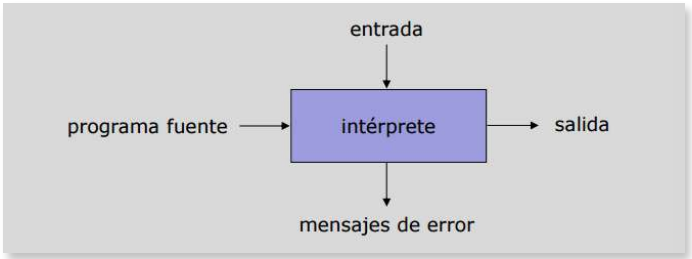


Por lo general, el código objeto es ejecutable en un ordenador, por lo que el lenguaje objeto es un lenguaje máquina.

Proceso de **compilación y ejecución**:



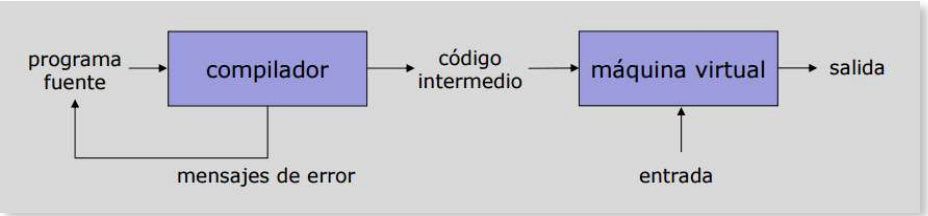
Un **intérprete** traduce y ejecuta, dando la sensación de ejecutar directamente cada una de las instrucciones del programa fuente, con las entradas proporcionadas por el usuario.



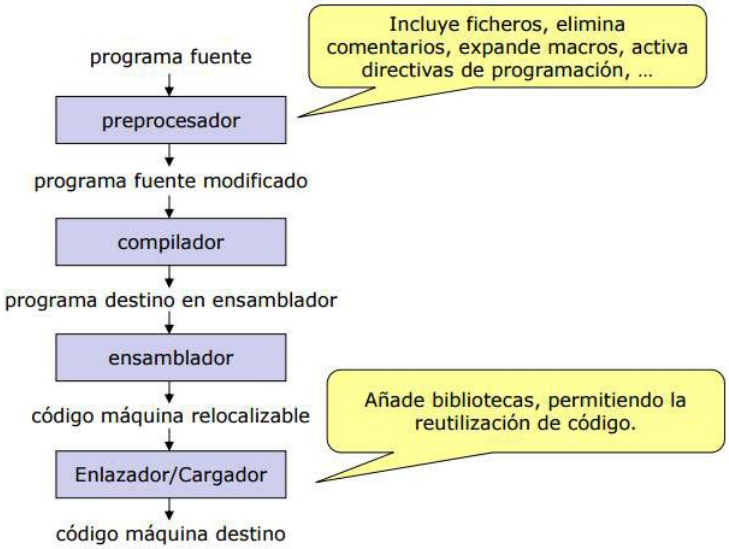
Compilador
- Compila 1 vez, ejecuta *n*
- Ejecución más rápida
- La gestión de errores abarca todo el programa

Intérprete
- Se traduce cada vez que se ejecuta
- Necesita menos memoria
- Se permite interacción con el código en tiempo de ejecución

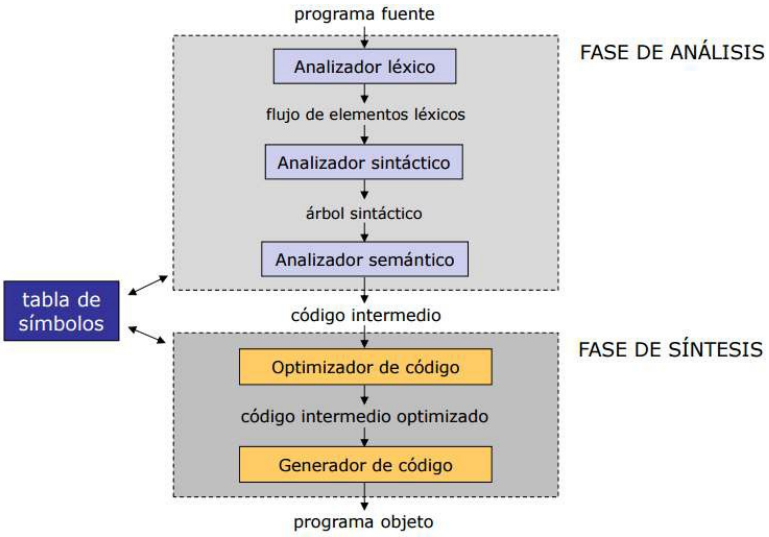
Un **compilador-intérprete** compila el programa fuente dando lugar a un programa escrito en un **lenguaje intermedio**. Después, una máquina virtual interpreta el programa. Combina las ventajas de compiladores e intérpretes.



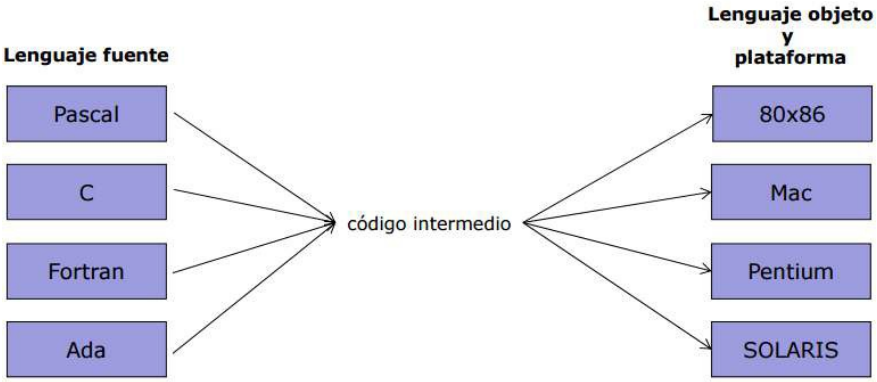
1.3. Proceso de traducción



1.4. Estructura de un compilador



Todos los lenguajes producen el mismo código intermedio, por lo que si trabajamos con un solo tipo de código intermedio sólo necesitamos una fase de análisis para cada lenguaje, y una fase de síntesis para cada plataforma.



1.4.1. ANÁLISIS LÉXICO

El **análisis léxico** lee el flujo de caracteres que constituyen el programa fuente y los agrupa en secuencias significativas o **componentes léxicos**. Mediante la **tabla de símbolos**, se almacena cada componente con su **lexema**.

1.4.2. ANÁLISIS SINTÁCTICO

El **análisis sintáctico** crea una representación intermedia en forma de árbol que describe la estructura gramatical del flujo de componentes léxicos. En un **árbol** sintáctico, cada nodo interior representa una operación, y los hijos del nodo representan sus argumentos.

1.4.3. ANÁLISIS SEMÁNTICO

El **análisis semántico** utiliza el árbol sintáctico y la tabla de símbolos para comprobar la consistencia semántica del código fuente.

1.4.4. GENERACIÓN DE CÓDIGO INTERMEDIO

El **código intermedio** es un código para una máquina abstracta. Debe ser fácil de producir y de traducir a código objeto. Una representación muy común es en forma de **código de tres direcciones**: secuencia de instrucciones, cada una de las cuales tiene como máximo 3 operandos.

1.4.5. OPTIMACIÓN DE CÓDIGO

La **optimización de código** trata de mejorar el código intermedio para que sea más rápido, más corto, consuma menos recursos, etc.

1.4.6. GENERACIÓN DE CÓDIGO

En la **generación de código** se traduce código intermedio a código máquina y se seleccionan las posiciones de memoria a utilizar.

1.4.7. TABLA DE SÍMBOLOS

La **tabla de símbolos** almacena los nombres de los elementos del programa fuente, junto con sus atributos.

1.5. CONSTRUCCIÓN DE COMPILADORES

En la **construcción de un compilador** se especifican 3 lenguajes:

1. **Lenguaje fuente**.
2. **Lenguaje objeto** y plataforma de ejecución.
3. **Lenguaje de implementación**, en el que está escrito el compilador.

1.5.1. DIAGRAMAS DE TOMBSTONE

Los **diagramas de Tombstone** son una herramienta visual que facilita la comprensión del proceso de diseño de compiladores e intérpretes. Hay 4 tipos:

1. **Compiladores.** Representa los 3 lenguajes de un compilador

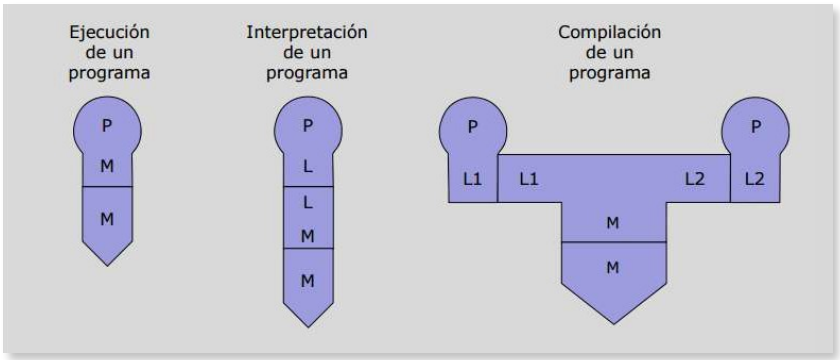
En este ejemplo tenemos un compilador de C a código máquina de PC. El compilador está escrito en C.

2. **Programas.** Representa el programa P escrito en lenguaje L

3. **Máquinas.** Representa la máquina o sistema operativo

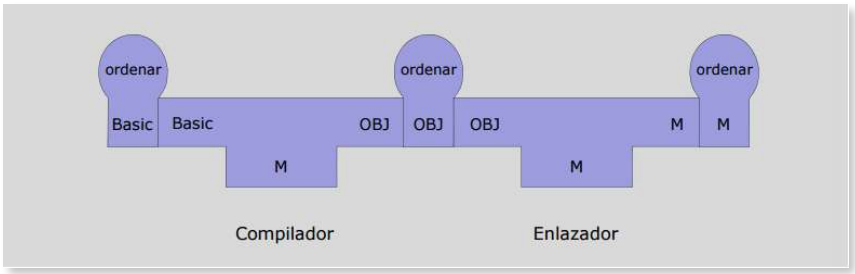
4. **Intérpretes.** Representa el lenguaje L escrito en M

Dos diagramas se pueden **unir** si en la unión los lenguajes son **iguales**.



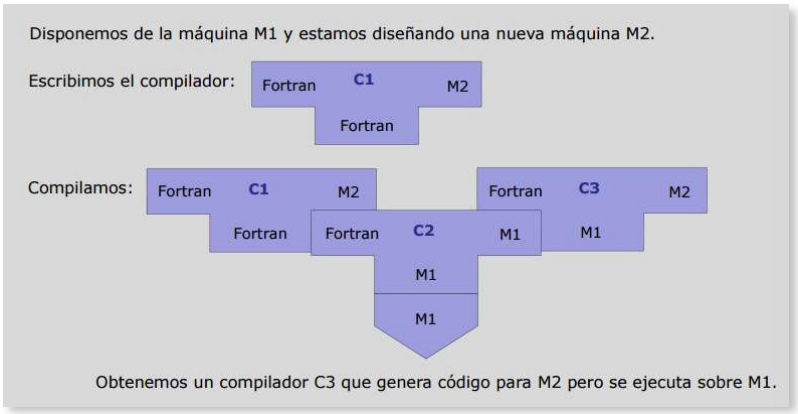
1.5.2. COMPILADOR ENLAZADOR

El **compilador-enlazador** es una técnica que divide la traducción en dos fases: **Traducción del lenguaje fuente a código intermedio** y **traducción del código intermedio a lenguaje objeto**. Puede utilizarse para desarrollar múltiples compiladores para múltiples plataformas.

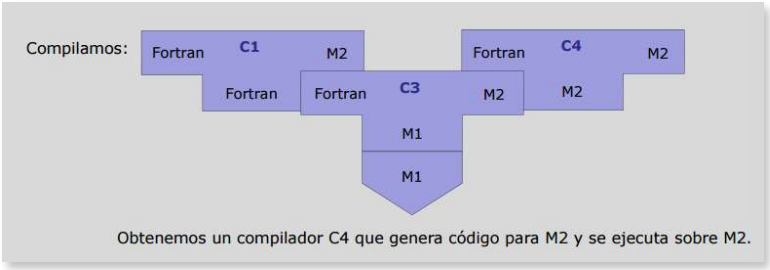


1.5.3. COMPILADOR CRUZADO

El **compilador cruzado** es una técnica que permite desarrollar compiladores para máquinas diferentes de la de desarrollo.

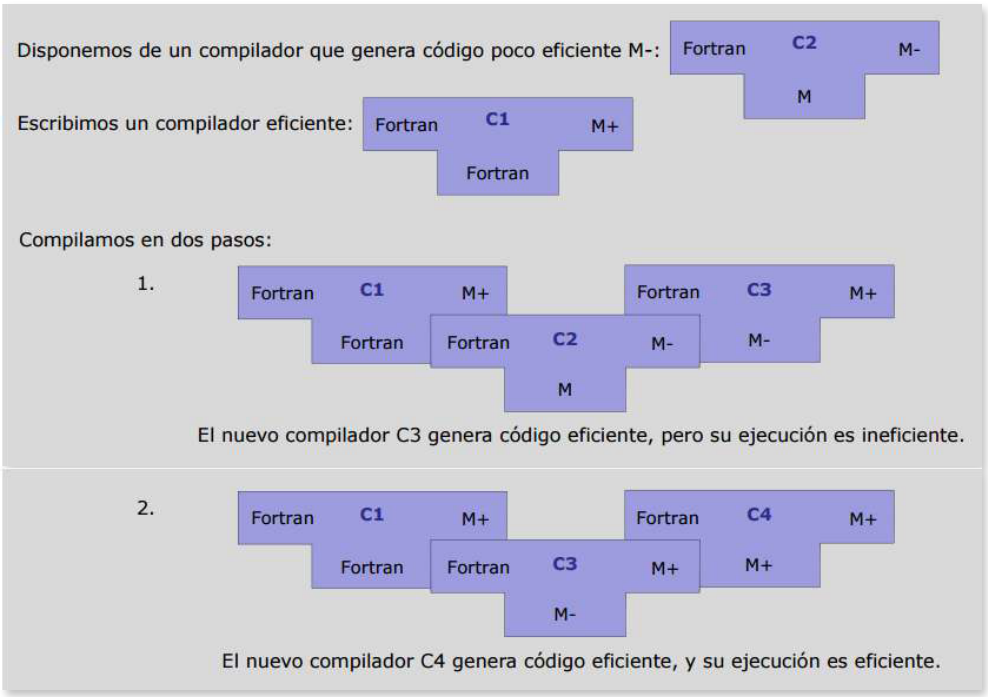


Si compilamos por segunda vez el compilador C1 con el compilador creado C3, obtendremos un compilador que genera código para M2 y se ejecuta sobre M2.



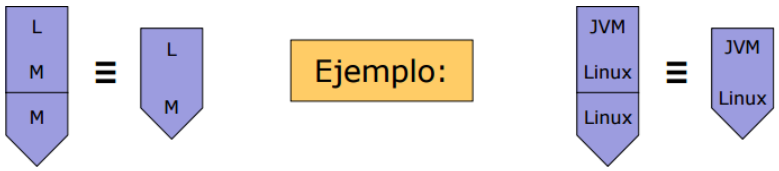
1.5.4. BOOTSTRAPPING

El **bootstrapping** es una técnica que se basa en el uso de múltiples etapas de compilación para mejorar la expresividad y rendimiento de un compilador. Una forma de bootstrapping consiste en construir el compilador de un lenguaje a partir de una versión reducida del propio lenguaje. De este modo, se obtienen sucesivas versiones que amplían y mejoran un lenguaje de programación determinado. Las mayores ventajas se obtienen cuando un compilador se escribe en el mismo lenguaje que compila. Un **autocompilador** es aquel que es capaz de compilar su propio código fuente.



1.5.5. COMPILADOR–INTÉRPRETE

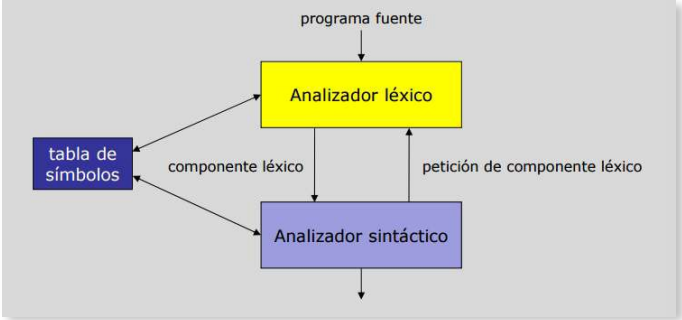
El **compilador-intérprete** resulta de la colaboración de un compilador y un intérprete, con el objetivo de generar un código intermedio fácilmente portable. La unión del intérprete y la plataforma de ejecución recibe el nombre de **máquina virtual**.



2. ANÁLISIS LÉXICO

2.1. ESTRUCTURA

Los analizadores sintáctico y léxico funcionan según el patrón productor–consumidor.



2.1.1. TAREAS DEL ANALIZADOR LÉXICO

El **analizador léxico** tiene las siguientes funciones:

1. Reconocer los componentes léxicos del lenguaje
2. Eliminar los espacios en blanco, caracteres de tabulación, saltos de línea y de página, y otros caracteres propios del dispositivo de entrada.
3. Eliminar los comentarios del programa fuente.
4. Reconocer los identificadores de variable, tipo, constantes, etc. y guardarlos en la tabla de símbolos.
5. Avisar de los errores léxicos detectados, y relacionar los mensajes de error con el lugar en el que aparecen en el programa fuente.

2.1.2. VENTAJAS DEL ANÁLISIS LÉXICO

Las ventajas de separar los análisis léxico y sintáctico son:

1. **Se simplifica el análisis.** Se evita la llegada de caracteres extraños al analizador sintáctico, que sólo ha de trabajar con la estructura del lenguaje. El diseño del compilador gana en claridad.
2. **Se mejora la eficiencia del compilador.** Pueden aplicarse técnicas específicas de mejora en cada fase.
3. **Se mejora la portabilidad del compilador.** Si se cambia el alfabeto de entrada, podremos cambiar el analizador léxico sin tener que tocar el sintáctico.

2.1.3. DISEÑO DEL ANÁLISIS LÉXICO

El **diseño del análisis léxico** se divide en etapas:

1. Especificación de los términos del análisis léxico mediante el uso de **expresiones regulares.**Diseño de un **autómata finito** que permita reconocer las expresiones regulares propuestas.
2. Realización de un **autómata finito determinista mínimo** que permita realizar un reconocimiento eficiente.
3. Diseño y realización de un **sistema de entrada**.
4. Diseño y realización de una **tabla de símbolos**.
5. Diseño y realización de una estrategia de **manejo de errores**.

2.2. ESPECIFICACIÓN DEL ANALIZADOR

2.2.1. TÉRMINOS DEL ANÁLISIS LÉXICO

El análisis léxico tiene los siguientes **términos**:

1. **Componente léxico:** Símbolo terminal de la gramática que define el lenguaje fuente.
2. **Patrón:** Expresión regular que define el conjunto de cadenas correspondientes a un componente léxico.
3. **Lexema:** Cadena de caracteres presente en el código fuente y que coincide con el patrón de un componente léxico.
4. **Atributos:** Acompañan a cada componente léxico encontrado y permiten su identificación y análisis posterior. En la práctica, un atributo apunta a una entrada de la tabla de símbolos.

2.2.2. EXPRESIONES REGULARES

Una **expresión regular r** permite representar patrones de caracteres. El conjunto de cadenas representado por r recibe el nombre de **lenguaje generado por r**, y se escribe L(r).

Para un alfabeto Σ diremos que:

1. El símbolo Ø (conjunto vacío) es una expresión regular y L(Ø)={ }.
2. El símbolo ε (palabra vacía) es una expresión regular y L(ε)={ε}.
3. Cualquier símbolo a $\in \Sigma$ es una expresión regular y L(a)={a}.

A partir de estas expresiones regulares básicas, pueden construirse expresiones regulares más complejas aplicando las siguientes **operaciones**:

1. **Concatenación:** Si `r` y `s` son expresiones regulares, entonces `rs` también lo es, y `L(rs)=L(r)L(s)`. Por ejemplo, si `L 1={00,1}` y `L 2={11,0}` entonces `L 1 L 2={0011,000,111,10}`. El lenguaje de la concatenación es la concatenación de los lenguajes.
2. **Unión:** Si `r` y `s` son expresiones regulares, entonces `r|s` también lo es, y `L(r|s)=L(r)U L(s)`. Por ejemplo, el lenguaje generado por `ab|c` es `L(ab|c)={ab,c}`.
3. **Cierre o clausura:** Si `r` es una expresión regular, entonces `r*` también lo es, y `L(r*)=L(r)*`. Por ejemplo, el lenguaje generado por `a*ba*` es `L(a*ba*)={b,ab,ba,aba,aab…}`. El cierre o clausura es la unión de las concatenaciones del lenguaje consigo mismo de `0` a `n` veces.

2.2.3. NOTACIÓN

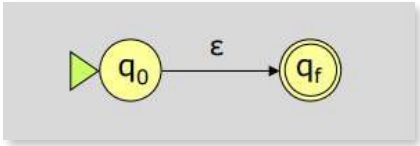
1. **Cero o un caso:** El operador unitario postfijo `?` significa "**cero o un caso de**". Así `r?` abrevia `r|ε`, y designa el lenguaje `L(r)U{ε}`.
2. **Uno o más casos:** El operador unitario postfijo + significa "**uno o más casos de**". Si `r` designa el lenguaje `L(r)`, entonces `r +` designa `(L(r)) +`. Se cumple que `r*=r +|ε` y `r +=rr*`.
3. **Clases de caracteres:** La notación `[abc]` designa la expresión regular `a|b|c`. `[a-z]` designa la expresión `a|b|…|z`.

2.3. AUTÓMATAS FINITOS

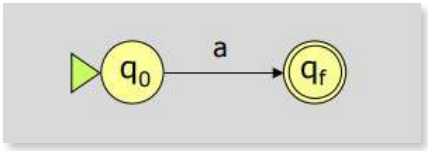
2.3.1. **DISEÑO DE UN AFN**

Para diseñar un AFN de un modo sistemático, se usará la **Construcción de Thompson**:

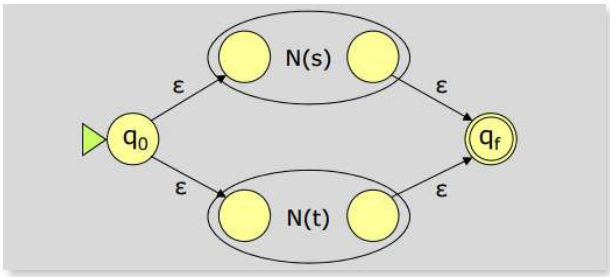
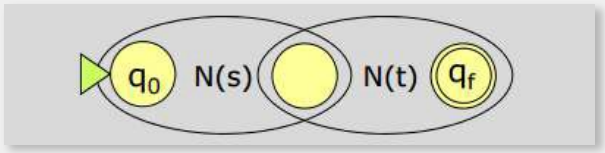
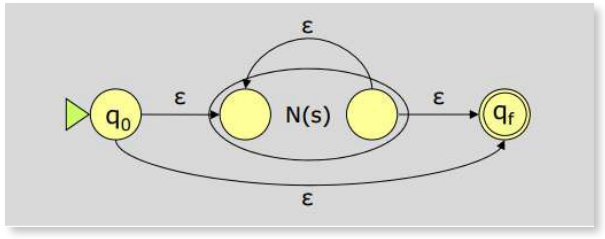
1. Para el símbolo ε, se construye el AFN siguiente:



2. Para el símbolo `a` ∈ Σ, se construye el AFN siguiente:



3. Supongamos que `N(s)` y `N(t)` son AFN para las expresiones regulares `s` y `t`:

Para la expresión regular `s|t` se construye el AFN `N(s|t)` siguiente:	
Para la expresión regular `st` se construye el AFN `N(st)` siguiente:	
Para la expresión regular `s*` se construye el AFN `N(s*)` siguiente:	

La aplicación sistemática de las reglas anteriores da lugar a un AFN N(r) que reconoce la expresión regular inicial `r`. Este AFN tiene las siguientes **propiedades**:

1. `N(r)` tiene un estado inicial y un estado final. Esta propiedad la satisfacen asimismo todos los AFN que lo componen.
2. `N(r)` tiene a lo sumo el doble de estados que dé símbolos y operadores en `r`: cada vez que se aplica una regla se crean a lo sumo dos nuevos estados.
3. Cada estado de `N(r)` tiene una transición saliente con un símbolo `a` ∈ Σ, o a lo sumo dos transiciones salientes con símbolos ε.

2.3.2. AFD EQUIVALENTE A UN AFN

Para obtener el AFD equivalente a un AFN dado, se usará el método de **construcción de subconjuntos**.

Sea el AFN $N=(Q^N,\Sigma,\delta^N,q^N_0,F^N)$, deseamos obtener un AFD equivalente $D=Q^D,\Sigma,\delta^D,q^D_0,F^D)$. Para ello, utilizaremos las tres operaciones siguientes:

4. **Cierre–ε(q)**, es el conjunto de estados del AFN N que se pueden alcanzar desde el estado $q \in Q^N$ con transiciones ε solamente.
5. **Cierre–ε(Q)**, es el conjunto de estados del AFN N que se pueden alcanzar desde los estados que pertenecen al conjunto $Q \subset Q^N$ con transiciones ε solamente.
6. **Mueve(Q, σ)**, es el conjunto de estados del AFN N a los cuales llega una transición desde los estados que pertenecen al conjunto $Q \subset Q^N$ con el símbolo $\sigma \in \Sigma$.

Para obtener un AFD D equivalente a un AFN N dado:

1. Antes de detectar el primer símbolo de entrada, N se puede encontrar en cualquiera de los estados del conjunto $cierre\text{-}\varepsilon(q^N)$. Por tanto, llamaremos q^D al conjunto de esos estados $q \in Q^N$ tales que $q \in cierre\text{-}\varepsilon(q^N)$.
2. Desde cualquier estado q^D_i hay una transición a un estado q^D_j con el símbolo $\sigma \in \Sigma$. Calculamos este estado q^D_j en dos pasos:
    a. Primero calculamos en el AFN N el conjunto $mueve(q^D_i,\sigma)$ (recordemos que $q^D_i \subset Q^N$).
    b. Calculamos su $cierre\text{-}\varepsilon$. En definitiva, $q^D_j=cierre\text{-}\varepsilon(mueve(q^D_i\ i,\sigma))$.
3. En el AFD D, un estado será final si contiene algún estado final del AFN N.

2.3.3. AFD MÍNIMO EQUIVALENTE

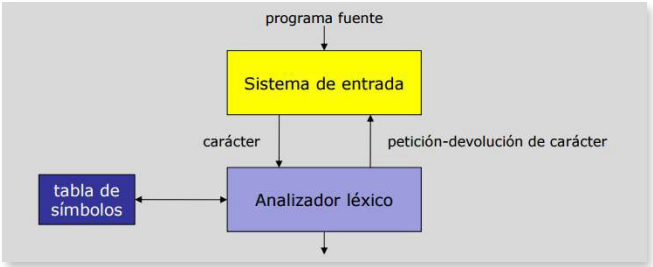
Un **analizador léxico** será más eficiente cuando menor sea el número de estados del AFD correspondiente. Para cualquier AFD existe un AFD mínimo equivalente. Para construir el AFD mínimo, se identifican pares de estados equivalentes.

2.3.4. CONSTRUCCIÓN DEL ANALIZADOR

En **analizador léxico** se construirá a partir de la agregación de AFD organizados según un orden conveniente. El último carácter leído será devuelto al flujo de entrada para ser leído en la siguiente iteración.

2.4. SISTEMA DE ENTRADA

El **sistema de entrada** es un conjunto de rutinas que interactúan con el sistema operativo para la lectura de datos del programa fuente. El sistema de entrada y el analizador léxico funcionan según el patrón productor–consumidor.



La separación del sistema de entrada supone una **mejora** en:

1. Eficiencia.
2. **Portabilidad.** El sistema de entrada es el único componente del compilador que se comunica con el SO. Para cambiar de plataforma sólo habría que cambiar el sistema de entrada

2.4.3. MEMORIA INTERMEDIA

El analizador léxico debe detectar el componente léxico con el lexema más largo posible. Se necesita poder leer caracteres de un modo anticipado. Para resolver este problema se debe incorporar una **memoria intermedia** de modo que:

1. Se pueda almacenar un bloque de caracteres en disco y apuntar el fragmento ya analizado.
2. En caso de devolución de caracteres al flujo de entrada se pueda mover un apuntador tantas posiciones como caracteres a devolver.

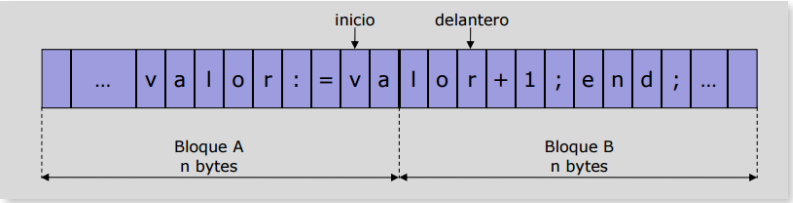
2.4.4. MÉTODOS DE GESTIÓN DE LA ENTRADA

Cualquier sistema de entrada debe **satisfacer**:

1. Ser lo más rápido posible.
2. Permitir un acceso eficiente a disco.
3. Hacer un uso eficiente de la memoria.
4. Soportar lexemas de longitud considerable.
5. Tener disponibles el lexema actual y el anterior.

2.4.5. MÉTODO DEL PAR DE MEMORIAS INTERMEDIAS

El **método del par de memorias intermedias** divide la memoria intermedia en dos mitades de `n` bytes cada una (`n` debe ser múltiplo de la longitud de la unidad de asignación).



Al principio, los apuntadores **inicio** y **delantero** apuntan al primer carácter de un lexema. A medida que el analizador pide caracteres, delantero se mueve hacia delante. Tras detectar un patrón, inicio avanza hasta la posición de delantero y se inicia el análisis del siguiente lexema.

Cada vez que se mueve delantero:

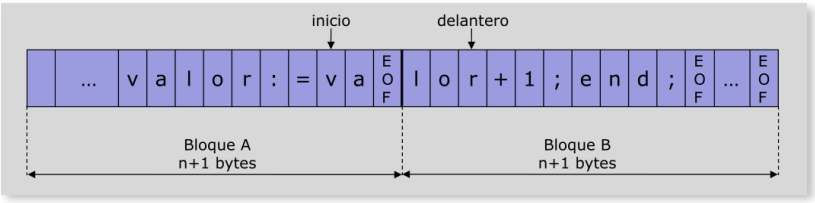
1. Se comprueba si se ha alcanzado el final de fichero.
2. Se comprueba si se ha alcanzado el final de un bloque. En caso de alcanzarse, se solicita un nuevo bloque al SO.

Limitaciones de este método:

1. El tamaño del lexema está limitado por n.
2. Es poco eficiente. Cada vez que delantero avanza hay que evaluar tres expresiones lógicas: Fin de fichero, fin de bloque A y fin de bloque B.

2.4.6. MÉTODO DEL CENTINELA

En el **método del centinela** se añade un byte más a cada bloque, en el que se guardará un carácter centinela (EOF). De este modo, se hace una sola comprobación lógica cada vez que avanza delantero. Si la comprobación es positiva, se analiza cuán de los tres casos es.



2.5. TABLA DE SÍMBOLOS

La **tabla de símbolos** es la estructura de datos utilizada por el compilador para gestionar los identificadores que aparecen en el programa fuente. Cuando el compilador encuentra un identificador, guarda en esta tabla toda la información que lo caracteriza. Cuando el identificador es referenciado en el programa, el compilador consulta la tabla de símbolos y obtiene la información que necesita. Una vez fuera del ámbito del identificador, se elimina de la tabla de símbolos.

2.5.1. PALABRAS RESERVADAS

Algunos lenguajes reservan algunas palabras que no pueden utilizarse como identificadores. **Pueden distinguirse:**

1. Definiéndolas mediante **expresiones regulares**, y asociándoles un componente léxico particular.
2. Mediante una **tabla de palabras clave**. Cada vez que se encuentra un lexema correspondiente a un identificador se busca en esta tabla.
3. Insertándolas al principio de la **tabla de símbolos**. Se gestionan en la misma tabla palabras clave e identificadores.

Durante la compilación:

1. El **analizador léxico**, cuando encuentra un identificador, comprueba que está en la tabla de símbolos. Si no lo está, crea una nueva entrada.
2. En **analizador sintáctico** añade información a los campos de los atributos. También puede crear nuevas entradas si se definen nuevos tipos de datos como palabras reservadas.
3. El **análisis semántico** debe acceder a la tabla para consultar los tipos de datos de los símbolos.
4. El **generador de código** puede:
    a. Leer el tipo de datos de una variable para la reserva de espacio
    b. Guardar la dirección memoria en la que se almacenará una variable

Cuando se encuentra un identificador, se consulta la tabla de símbolos:

1. Si se encuentra entre las palabras reservadas, se devuelve su componente léxico al analizador sintáctico.
2. Si se encuentra después de las palabras reservadas, es un identificador previamente encontrado.
3. Si no se encuentra en la tabla de símbolos, se añade como un nuevo identificador.

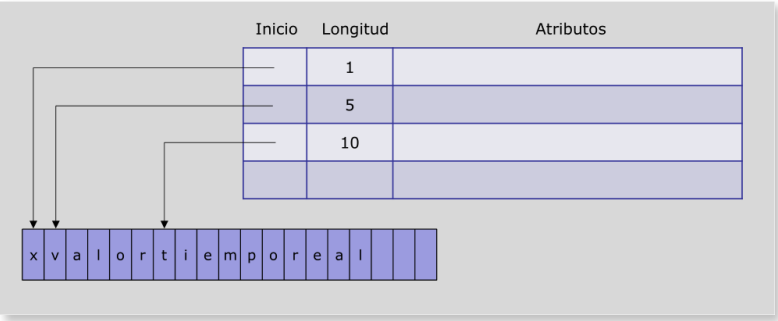
2.5.2. ESTRUCTURA DE LA TABLA DE SÍMBOLOS

La **tabla de símbolos** se estructura en un conjunto de registros, cuya longitud suele ser fija, conteniendo el lexema encontrado y un conjunto de atributos para su componente léxico. Hay dos **formas de almacenamiento:**

1. **Estructura interna.**

2. **Estructura externa.** Esta estructura no exige una longitud fija para los identificadores, por lo que se aprovecha mejor el espacio de almacenamiento.

Atributos comunes en la tabla de símbolos:



1. **Número de línea** donde aparece el símbolo.
2. **Tipo de dato** representado por el símbolo, o apuntador a la estructura de datos correspondiente al tipo. El encargado de llenar este campo es el analizador sintáctico.
3. **Dirección de memoria** en la que se guardará el valor correspondiente al símbolo durante la ejecución del programa. El generador de código rellena este campo y sustituye en el código objeto el identificador por su dirección de memoria.

2.5.3. OPERACIONES

La tabla de símbolos funciona como una base de datos cuya clave de búsqueda es el lexema de un símbolo. Las **operaciones** que se ejecutan sobre la tabla son:

1. **Buscar** un lexema y el contenido de sus atributos.
2. **Insertar** un nuevo registro previa comprobación de su no existencia.
3. Modificar la información contenida en un registro. Además, en los lenguajes con estructura de bloque: **Nuevo bloque** y **Fin de bloque**.

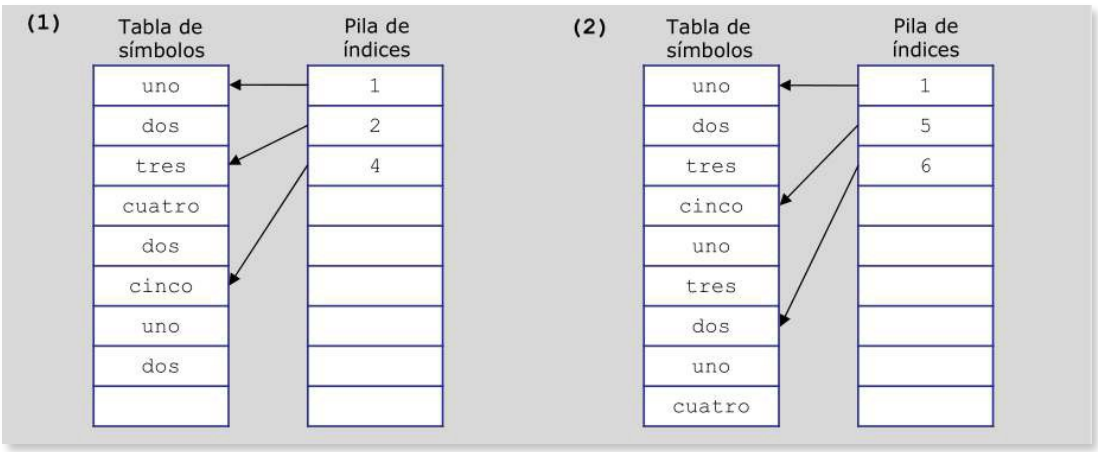
2.5.4. ORGANIZACIÓN DE LA TABLA

Las tablas de símbolos se organizan de dos maneras:

1. **Tablas no ordenadas.** Generadas con vectores o listas. Poco eficientes.
2. **Tablas ordenadas.** Permiten definir el **tipo diccionario**. Utilizan estructuras de tipo vector o lista ordenada, árboles binarios, árboles equilibrados, tablas hash, etc.

2.5.5. TABLAS DE SÍMBOLOS NO ORDENADAS

Las **tablas de símbolos no ordenadas** usan un vector o una lista. Se añade una pila auxiliar de apuntadores de índice de bloque, para marcar el comienzo de los símbolos que corresponden a un bloque. Al terminar un bloque, se eliminan todos los símbolos desde el siguiente al apuntado hasta el final de la tabla de símbolos.

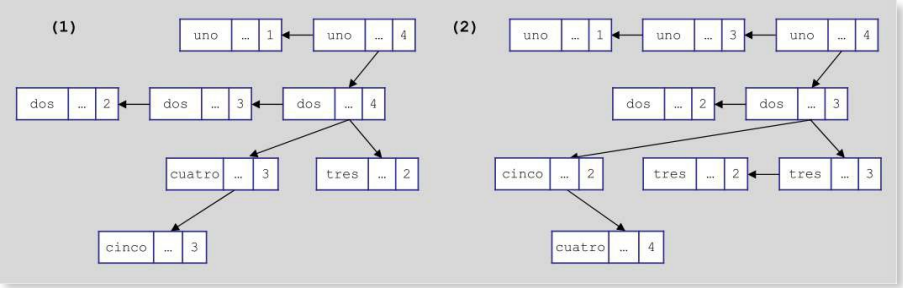


Admite las siguientes **operaciones:**

1. **Inserción.** Cuando se encuentra la declaración de un nuevo símbolo, se verifica que no se encuentra en el último bloque. Si no está, se inserta en la última posición de la tabla, si está, se devuelve un error.
2. **Búsqueda.** La búsqueda se hace desde el final de la tabla hacia el principio. Si se encuentra, se corresponde a la declaración realizada en el bloque más próximo.
3. **Nuevo bloque.** Cuando empieza un bloque se añade en la pila un apuntador al último símbolo de la tabla.
4. **Fin de bloque.** Cuando termina un bloque, se eliminan todos los símbolos del mismo desde el siguiente al de inicio, apuntado desde la pila. Luego se elimina el índice de la pila.

2.5.6. TABLA DE SÍMBOLOS CON ESTRUCTURA DE ÁRBOL

Se mejora la eficiencia estructurando la tabla de símbolos mediante un **árbol binario ordenado**, y añadiendo un campo a cada registro que indique el nivel al que pertenece el símbolo. Para evitar duplicidades, se usan listas encadenadas.

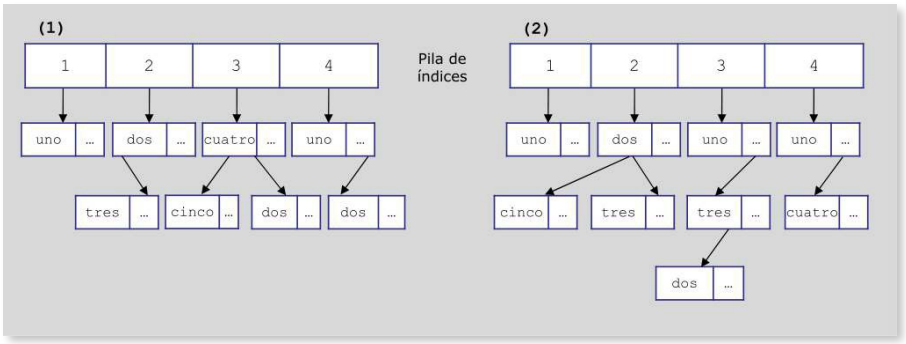


Admite las siguientes **operaciones:**

1. **Inserción.** Para insertar un nuevo símbolo, se busca la posición que le corresponde. Si ya hay un registro con el mismo lexema, se comprueba que el campo nivel no tiene el mismo valor que el bloque activo. Si fuese así, habría que indicar un error. En caso contrario, se añade un nuevo nodo a la lista del primero. Si no hay registro, se inserta directamente en el árbol.
2. **Búsqueda.** La propia del árbol binario.
3. **Nuevo bloque.** Se incrementa en uno el campo de nivel.
4. **Fin de bloque.** Se eliminan los símbolos del bloque:
    a. Se localizan los registros del árbol del bloque activo.
    b. Se borran.
    c. Se decrementa en 1 la variable con el número de nivel.

2.5.7. TABLA DE SÍMBOLOS CON ESTRUCTURA DE BOSQUE

La **tabla de símbolos con estructura de** bosque utiliza un árbol para cada bloque del programa, y una pila de índices de nivel que apuntan a la raíz del árbol de nivel.



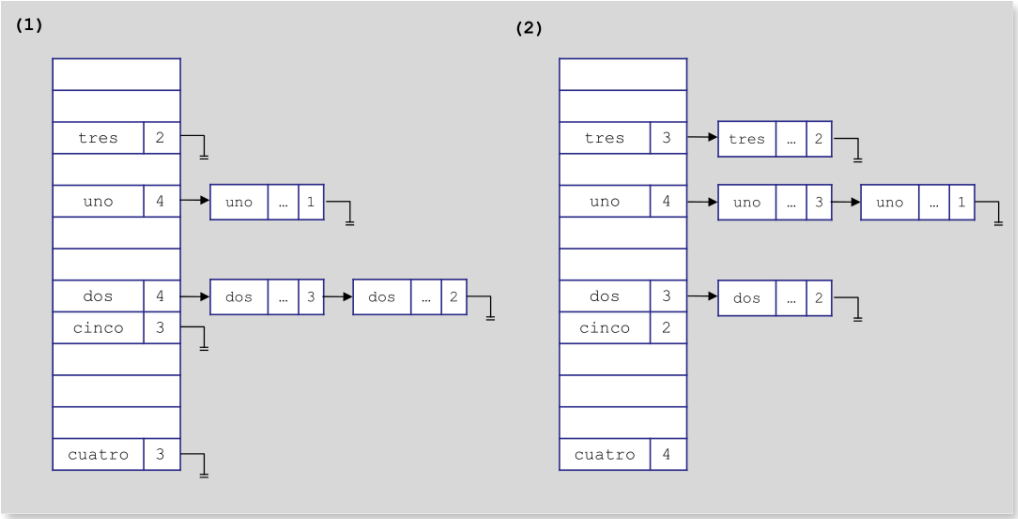
Admite las siguientes **operaciones**

1. **Inserción.** Consiste en hacer una inserción normal en el árbol del bloque activo.
2. **Búsqueda.** Primero se busca el lexema en el árbol del bloque activo. Si no se encuentra, se busca en el árbol del bloque anterior, y así sucesivamente.
3. **Nuevo bloque.** Cuando empieza la compilación de un nuevo nivel, se crea un nuevo elemento en la pila de índices y una nueva estructura de tipo árbol.
4. **Fin de bloque.** Cuando se termina de compilar un bloque se destruye el árbol asociado y se elimina el último puntero de la pila de índices.

2.5.8. TABLAS HASH

Una **tabla hash** aplica una función matemática al lexema para determinar la posición de la tabla que le corresponde. El inconveniente de las tablas hash es que pueden producirse **colisiones** cuando dos lexemas quieren ocupar la misma solución. Existen dos **soluciones**:

1. **Tablas hash cerradas.** Cuando se produce una colisión, se usa una técnica que permita acceder a una posición vacía.
2. **Tablas hash abiertas.** Se utiliza una lista encadenada para resolver las colisiones.



Admite las siguientes **operaciones:**

1. **Inserción.** Como en una tabla hash normal. Se suelen usar listas de desbordamiento, colocando los símbolos más recientes al principio.
2. **Búsqueda.** Como en una tabla hash normal. En caso de colisión, de busca en la lista de desbordamiento.
3. **Nuevo bloque.** Se almacena el cambio de bloque activo.
4. **Fin de bloque.** Se borran todos los registros correspondientes a ese bloque. Implica recorrer prácticamente toda la tabla. Después se cambia el bloque activo.

2.5.9. ANÁLISIS DE COMPLEJIDAD TEMPORAL

	Búsqueda	Inserción
Lista no ordenada	O(n)	O(n)
Lista ordenada Árbol AVL	O(log n)	O(n)
Tabla hash	O(log n)	O(log n)
	O(1)	O(1)

2.6. TRATAMIENTO DE ERRORES

Cuando en la compilación **se detecta un error**:

1. Se debe mostrar un **mensaje claro y exacto**, que permita al programador encontrarlo y corregirlo.
2. Debe **recuperarse el error** e ir a un estado que permita continuar analizando el programa. Se debe evitar la cascada de errores o pasar por alto otros.
3. **No debe retrasar** excesivamente el procesamiento de programas correctos.

La **recuperación de un error** puede hacerse adoptando diversas medidas:

1. **Ignorar** los caracteres inválidos hasta formar un componente léxico correcto.
2. **Eliminar** caracteres que dan lugar a error.
3. **Intentar corregir el error.** Puede ser peligroso.
    a. **Insertar** los caracteres que puedan faltar.
    b. **Reemplazar** un carácter presuntamente incorrecto por uno correcto.
    c. **Intercambiar** caracteres adyacentes.

3. ANÁLISIS SINTÁCTICO

3.1. NOCIONES GENERALES

En el **análisis sintáctico** se comprueba que cada programa escrito en código fuente obedece a las reglas de la gramática.

3.1.1. GRAMÁTICAS INDEPENDIENTES DEL CONTEXTO

Una **gramática independiente del contexto** e define como una 4-tupla G=(Σ,V,S,P), siendo:

1. **Σ** un alfabeto o conjunto de **símbolos terminales**.
2. **V** un conjunto de variables o **símbolos no terminales**.
3. **S** una variable denominada **símbolo inicial**.
4. **P** una colección de **reglas de sustitución**, llamadas reglas de producción de la forma $A \rightarrow \alpha$, donde $A \in V$, y $\alpha \in (V \cup \Sigma)^*$.

La máquina apropiada para el reconocimiento de gramáticas independientes del contexto es el **autómata a pila**. Se introduce el símbolo **$** para indicar el **final de cadena**.

3.1.2. TIPOS DE ANALIZADORES SINTÁCTICOS

Un **analizador sintáctico** analiza el código fuente como una secuencia de componentes léxicos, y construye la representación interna en forma de árbol sintáctico. Hay dos tipos de analizadores:

1. **Descendentes.** Parten de la raíz del árbol. Las reglas se aplican sustituyendo la parte izquierda de las mismas por su parte derecha.
2. **Ascendentes.** Parten de las hojas del árbol. Las reglas se aplican reduciendo la parte derecha de las mismas por su parte izquierda.

3.1.3. FORMA DE BNF EXTENDIDA

La **forma de BNF extendida** es la forma más común para formalizar gramáticas. Los metasímbolos más comunes que utiliza son:

1. **::=** Definiciones
2. | Disyunción
3. **<>** Símbolos no terminales
4. **[ ]** Símbolos opcionales. Pueden aparecer cero o una veces.
5. **{ }** Repetición. Símbolos que pueden aparecer cero, una o más veces.
6. **“** Se utiliza para distinguir metasímbolos de terminales.

3.1.4. FORMA DE BNF VISUAL

1. Los símbolos terminales se marcan en negrita.
2. Se eliminan los símbolos <> de los no terminales.
3. Se utiliza el símbolo → en lugar de **::=**.

3.2. DISEÑO DE UNA GRAMÁTICA

Cuando se diseña un lenguaje de programación, se debe diseñar una gramática que describa las características del lenguaje. Algunos aspectos que deben ser discutidos son:

1. **Recursividad.** Permite reducir el número de reglas sintácticas y generar un número infinito de programas mediante una gramática finita. Una gramática es **recursiva** si al reescribir un no terminal, éste vuelve a aparecer en una o más derivaciones $A \Rightarrow^{+} \alpha A \beta$.
2. **Ambigüedad.** Proporciona gramáticas más intuitivas, pero permite generar códigos objeto diferentes a partir del mismo código fuente. Una gramática es **ambigua** si genera al menos una sentencia mediante dos o más árboles de derivación diferentes. Se debe evitar.
3. **Factorización por la izquierda.** Durante el diseño de la gramática, pueden aparecer dos producciones de un mismo no terminal que empiezan igual por lo que, cuando se llegue a este no terminal, no se sabrá cuál elegir. Una solución es reescribir estas reglas, retrasando la decisión hasta que se haya visto suficiente (**factorización por la izquierda**). En general, si la gramática tiene una producción de la forma $A \rightarrow \alpha\beta_1 \mid \alpha\beta_2 \mid \ldots \mid \alpha\beta_n \mid \gamma$, se puede realizar la sustitución siguiente:

   $$A \rightarrow \alpha A' \mid \gamma$$
   $$A' \rightarrow \beta_1 \mid \beta_2 \mid \ldots \mid \beta_n$$

   A continuación, se comprueba si $\beta_1 \mid \beta_2 \mid \ldots \mid \beta_n$ también se puede factorizar.
4. **Asociatividad.** Se deben proporcionar reglas de asociatividad para las operaciones del lenguaje. Tipos de asociatividad:
   a. **Asociatividad por la derecha.** Las operaciones se hacen de derecha a izquierda (Asignación)
   b. **Asociatividad por la izquierda.** Las operaciones se hacen de izquierda a derecha (Operaciones aritméticas)

   La asociatividad se puede incorporar a una gramática mediante la recursividad.
   Se distingue entre:
   a. **Operador asociativo por la derecha.** La regla sintáctica en la que aparece el operador se hace recursiva por la derecha.
   b. **Operador asociativo por la izquierda.** La regla sintáctica en la que aparece el operador se hace recursiva por la izquierda.
5. **Precedencia.** Determina el orden en el que se realizarán las operaciones del lenguaje. Cuando en una expresión intervienen varios operadores, se puede producir ambigüedad al construir el árbol de derivación. La precedencia permite especificar un **orden relativo de evaluación** de unos operadores respecto a otros. Formas de incorporar precedencia:
   a. Usar una variable sintáctica para cada nivel de precedencia.
   b. Usar una producción para cada operador.

   Un operador tendrá menos precedencia cuanto más cerca esté su regla de derivación de la regla inicial de la gramática.

3.3. ANÁLISIS SINTÁCTICO DESCENDENTE

En el caso más simple, el **análisis sintáctico descendente** es completamente recursivo:

1. **Avanzamos** cuando reescribimos un no terminal.
2. **Retrocedemos** cuando se llega a un punto muerto y es necesario probar otra regla de sustitución.

Para gestionar este proceso, se unas una **pila**, donde se almacenan los símbolos de cada sustitución. Se intenta emparejar el símbolo de la cima de la pila con la entrada actual. Si al final del proceso, la pila y la entrada están vacías, entonces la sentencia pertenece al lenguaje.

Los métodos del ASD tienen algunos inconvenientes:

1. Son muy lentos debido a los retrocesos.
2. No se sabe si la entrada pertenece o no al lenguaje hasta analizar todas las posibilidades. Si no pertenece, no podremos indicar dónde está el error.
3. Si se genera código durante el análisis, en cada retroceso hay que eliminar parte del mismo.

3.3.1. RECURSIVIDAD POR LA IZQUIERDA

Las **reglas recursivas por la izquierda** pueden hacer entrar al ASD en un bucle infinito, por lo que deben ser eliminadas.

Sea una gramática recursiva:

$$A \rightarrow A\alpha_1 \mid A\alpha_2 \mid \dots \mid A\alpha_n \mid \beta_1 \mid \beta_2 \mid \dots \mid \beta_m$$

Una gramática no recursiva por la izquierda equivalente será: $A \rightarrow \beta_1 A' \mid \beta_2 A' \mid \dots \mid \beta_m A'$

$$A' \rightarrow \alpha_1 A' \mid \alpha_2 A' \mid \dots \mid \alpha_n A' \mid \varepsilon$$

El método anterior elimina la recursividad inmediata, esto es, en la misma derivación. Sin embargo, la recursividad puede aparecer en derivaciones posteriores, en cuyo caso debemos encontrar el elemento conflictivo y sustituirlo por su definición, para luego aplicar el método anterior.

3.3.2. ANALIZADOR SINTÁCTICO PREDICTIVO

El analizador sintáctico predictivo es capaz de evitar los retrocesos leyendo k componentes léxicos por anticipado. Para ello, debe utilizarse un subconjunto de gramáticas denominadas **LL(K)**:

- **L(left):** La entrada se lee de izquierda a derecha
- **L(left):** Se sustituye el no terminal situado a la izquierda
- **(k):** Se leen k componentes léxicos por anticipado

En concreto, nos centraremos en gramáticas de tipo `LL(1)`, con un solo componente léxico analizado por anticipado. Una gramática debe cumplir las siguientes condiciones necesarias pero no suficientes:

1. No puede ser recursiva por la izquierda
2. Dos reglas de sustitución de un mismo no terminal no pueden dar lugar al mismo componente léxico inicial.

Si encontramos una gramática no `LL(1)`, la modificaremos:

1. Eliminando la recursividad por la izquierda.
2. Sacando factor común por la izquierda.

3.3.3. CONJUNTOS DE PREDICCIÓN

Se definen los conjuntos **primeros** y **siguientes** que ayudarán en el proceso de análisis predictivo.

☑ PRIMEROS

Sea X un símbolo gramatical $(X \in (V \cup \Sigma))$, PRIMEROS(X) es el conjunto de terminales (incluyendo ε) que aparecen al inicio de las cadenas derivables de X en ninguno o más pasos.

$$PRIMEROS(X) = \{v \mid X \Rightarrow^* v\beta;\ v \in \Sigma,\ \beta \in \Sigma^*\}$$

Para calcular PRIMEROS(X):

1. Si X es un símbolo terminal a, PRIMEROS(X)=a
2. Si $(X \rightarrow \varepsilon) \in P$, añadimos ε a PRIMEROS(X).
3. Si X es un no terminal, y $X \rightarrow Y_1\ Y_2\ \dots\ Y_k$ es una regla de sustitución, añadimos PRIMEROS(Y_j) a PRIMEROS(X), si PRIMEROS(Y_j) es un terminal, y para todo $i=1,2,\dots,j-1$, $Y_i \Rightarrow^* \varepsilon$.
4. 4. Si X es un no terminal, $X \rightarrow Y_1\ Y_2\ \dots Y_k$ es una regla de sustitución, y $Y_j \Rightarrow^* \varepsilon\ \forall\ j \in \{1, 2, \dots, k\}$, añadimos ε a PRIMEROS(X).

Para calcular PRIMEROS($X_1\ X_2\ \dots X_n$):

1. Añadimos todos los símbolos ≠ de ε de PRIMEROS(X_1).
2. Si ε está en PRIMEROS(X_1), añadir también los símbolos distintos de ε de PRIMEROS(X_2).
3. Si ε está en PRIMEROS(X_2), añadir también los símbolos distintos de ε de PRIMEROS(X_3). Y así sucesivamente.
4. Por último, añadir ε a PRIMEROS($X_1\ X_2\ \dots X_n$) si PRIMEROS(X_i) contiene a ε, $\forall\ i \in \{1,2,\dots,n\}$.

☑ SIGUIENTES

Sea A un símbolo **no terminal**. SIGUIENTES(A) es el conjunto de terminales que pueden aparecer inmediatamente a la **derecha de A** en alguna sustitución.

$$SIGUIENTES(A) = \{v \mid S \Rightarrow^+ \alpha A v \beta;\ v \in \Sigma,\ \alpha, \beta \in \Sigma^*\}$$

Si A aparece al final de una cadena de símbolos, entonces diremos que el símbolo de fin de cadena $ ∈ SIGUIENTES(A).

Para calcular SIGUIENTES(A):

1. Si A es S, añadir **$** a SIGUIENTES(S).
2. Para cada regla $B \rightarrow \alpha A \beta$, añadir **PRIMEROS(β)-ε** a SIGUIENTES(A).
3. Para cada regla $B \rightarrow \alpha A \beta$, donde PRIMEROS(β) contiene a ε $(\beta \Rightarrow^* \varepsilon)$, añadir todos los símbolos de **SIGUIENTES(B)** a SIGUIENTES(A).
4. Para cada regla $B \rightarrow \alpha A$, añadir **SIGUIENTES(B)** a SIGUIENTES(A).

3.3.4. TABLA DE ANÁLISIS SINTÁCTICO

Mediante los **conjuntos de predicción** podemos construir una tabla que, a partir del no terminal a expandir y del componente léxico actual, indique qué regla se debe aplicar. En la **tabla de análisis sintáctico** las filas son los símbolos no terminales de la gramática, y sus columnas son los componentes léxicos, y el símbolo de fin de cadena (la cadena vacía no puede aparecer en las columnas). En las celdas de la tabla aparecen las reglas de la gramática. Las celdas vacías se corresponden con un error en el análisis.

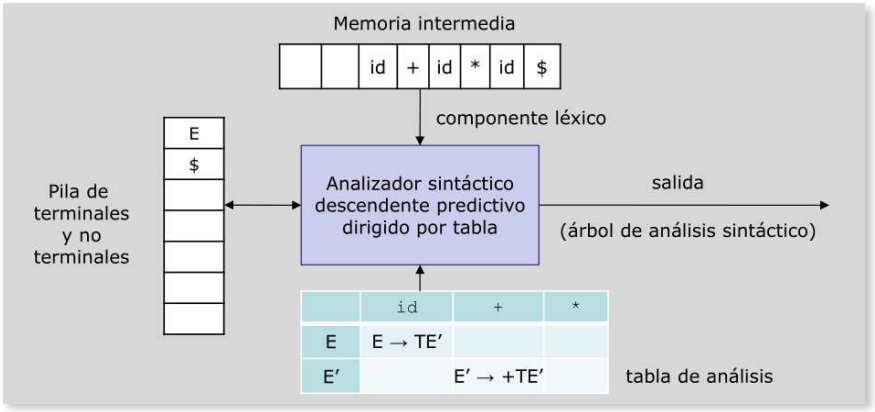
Las celdas de la tabla `T[A, a]` se rellenan siguiendo el siguiente procedimiento para cada regla `A → α`:

1. Para cada terminal `a ∈ PRIMEROS(α)`, añadir `A → α` en `T[A, a]`.
2. 2. Si `ε ∈ PRIMEROS(α)`, para cada terminal `b ∈ SIGUIENTES(A)`, añadir `A → α` en `T[A, b]`.

Una gramática es `LL(1)` si en la tabla de análisis sintáctico aparece como máximo una regla en cada celda.

3.3.5. PROCEDIMIENTO DE ANÁLISIS

1. Inicializar la pila con el símbolo `S`, y añadir `$` a continuación y al final de la cadena de entrada.
2. Extraer el símbolo `X` de la cima de la pila y el siguiente símbolo `a` de la entrada.
   a. Si `X = a = $` **aceptar** la cadena y salir.
   b. Si `X = a ≠ $`, **extraer** el elemento de la pila y **avanzar** una posición en la cadena de entrada.
   c. Si `X ≠ a`, `t T(X, a)` está vacía, entonces **error**.
   d. Si `X ≠ a`, `t` en `T(X, a)` tenemos `X → X₁ X₂ … Xₙ` **extraer** `X` de la pila e **insertar** `X₁ X₂ … Xₙ`.
   e. Volver a 2.



3.3.6. GESTIÓN DE ERRORES

Tipos de errores en el ASD:

- En la parte superior de la pila hay un terminal que no se corresponde con el componente léxico actual. El compilador deberá señalar el error, indicando qué es lo que esperaba encontrarse.
- En la parte superior de la pila hay un no terminal, y la celda de la tabla correspondiente al componente léxico actual está vacía. El compilador debe señalar un error, conforme se esperaba uno de los componentes léxicos del conjunto de celdas del no terminal que contienen algún dato.

Conjuntos de sincronización

Al detectar un error, el compilador debe continuar el análisis, pudiendo aplicar para ello la técnica de conjuntos de sincronización:

Al detectar un error, se entra en un estado de error en el que se continúa analizando la entrada, descartando los componentes léxicos que aparecen hasta encontrar uno que pertenezca al conjunto de sincronización. En este momento, se quita el símbolo de encima de la pila, se abandona el estado de error y se prosigue el análisis.

La efectividad de este método depende de la elección del conjunto de símbolos de sincronización.

3.4. ANÁLISIS SINTÁCTICO ASCENDENTE

El **análisis sintáctico ascendente** parte de una cadena de entrada, y busca hacerla corresponder con las partes derechas de la gramática. Cuando encuentra una, la sustituye (reduce) por su parte izquierda, y crea un nuevo nodo del árbol sintáctico. Se continúa hasta alcanzar `S` o encontrar un error. La cadena se procesa de izquierda a derecha, y en cada paso se sustituyen uno o más símbolos por una única variable.

Los algoritmos de ASA utilizan una **pila** para el análisis y realizan dos operaciones básicas:

1. **Desplazamiento.** Lleva el siguiente símbolo de entrada a la cima de la pila.
2. **Reducción.** Extrae tantos símbolos de la pila como símbolos tiene la parte derecha de la regla de sustitución, por la cual se quiere reducir, y los sustituye por el símbolo no terminal de la parte izquierda de la regla.

3.4.1. GRAMÁTICAS `LR(k)`

En el análisis sintáctico ascendente por desplazamiento–reducción se usan las **gramáticas `LR(k)`**:

- **`L (left):`** La entrada se lee de izquierda a derecha
- **`R (right):`** Se construyen derivaciones canónicas por la derecha
- **`(k):`** Se leen k componentes léxicos por adelantado. Para cada gramática `LR(k)` con `k > 1` existe una gramática `LR(1)` equivalente..

Las gramáticas `LR(1)` **no son ambiguas** y describir más lenguajes que las `LL(1)`, *englobando* a éstas.

Tipos de gramáticas `LR(1)`:

1. **`SLR(1).`** Presentan la construcción más sencilla.
2. **`LR(1).`** Permiten describir un rango mayor de gramáticas, pero la construcción es más compleja.
3. **`LALR(1).`** Compromiso entre las dos anteriores.

3.4.2. PROCEDIMIENTO DE ANÁLISIS

Para determinar cuándo desplazar y cuándo reducir, se usa un procedimiento basado en un **AFD**, que nos dirá si una entrada pertenece al lenguaje generado por la gramática. Sus estados son conjuntos de elementos que representan situaciones equivalentes en el análisis de una entrada. Se utilizará una función denominada **`Ir_a`** para simular las transiciones del autómata para cada uno de los símbolos de la gramática. Para cada gramática `G`, es preciso crear su **gramática aumentada**, resultado de añadir la regla **`S'→S$`**. Cuando se vaya a reducir mediante esta regla, se acepta.

3.4.3. GRAMÁTICAS `SLR(1)`

Elementos

Un **`elemento`** es una regla simple que contiene una marca en algún lugar de la parte derecha, que separa los símbolos de la regla que han sido reconocidos de los pendientes de reconocer. Para las reglas de sustitución `A → ε` sólo se obtiene el elemento `A → •`.

Clausura

La función **`clausura()`** se aplica a un conjunto de elementos `I` y devuelve un nuevo conjunto de elementos que son equivalentes respecto a dónde ha llegado el análisis.

El conjunto de elementos que devuelve la `clausura(I)` se construye:

1. Todo elemento de `I` se añade a la `clausura(I)`.
2. Si `A→α•Bβ` está en `clausura(I)` y `B→γ` es una regla de la gramática, entonces añadir `B→•γ` a `clausura(I)`. Aplicar esta regla hasta que no se puedan añadir nuevos elementos a `clausura(I)`.

`Clausura(I)` agrupa a todos los elementos que describen lo que se espera encontrar a partir del momento actual del análisis.

Ir_a(I,x)

La función **`Ir_a(I,X)`** se aplica a un conjunto de elementos `I`, y a un símbolo de la gramática $X \in (V \cup \Sigma)$, y devuelve un nuevo conjunto de elementos.

`Ir_a(X,I)` se define como la clausura del conjunto de todos los elementos $A \rightarrow \alpha X \bullet \beta$ tales que $A \rightarrow \alpha \bullet X \beta$ está en `I`. El conjunto que devuelve `Ir_a(I,X)` está formado por aquellos elementos de `I` que después han reconocido el símbolo `X`.

Colección canónica LR(0)

Llamamos **colección canónica `LR(0)`** al conjunto de todos los posibles conjuntos de elementos de una gramática.

Para generar la colección canónica de una gramática:

1. Creamos su gramática aumentada.
2. El primer conjunto de elementos es I_0=`clausura(S'→•S$)`.
3. Se calculan todos los posibles conjuntos resultado de aplicar `Ir_a` de nuevo a I_1, I_2, … hasta no obtener nuevos conjuntos.

Si consideramos cada conjunto I_j como un estado de un autómata finito, `Ir_a` nos proporciona su **tabla de transiciones**.

A continuación, se construye una **tabla de acciones** que nos permita saber cuándo el analizador sintáctico tiene que *reducir*, y cuándo tiene que *desplazar*.

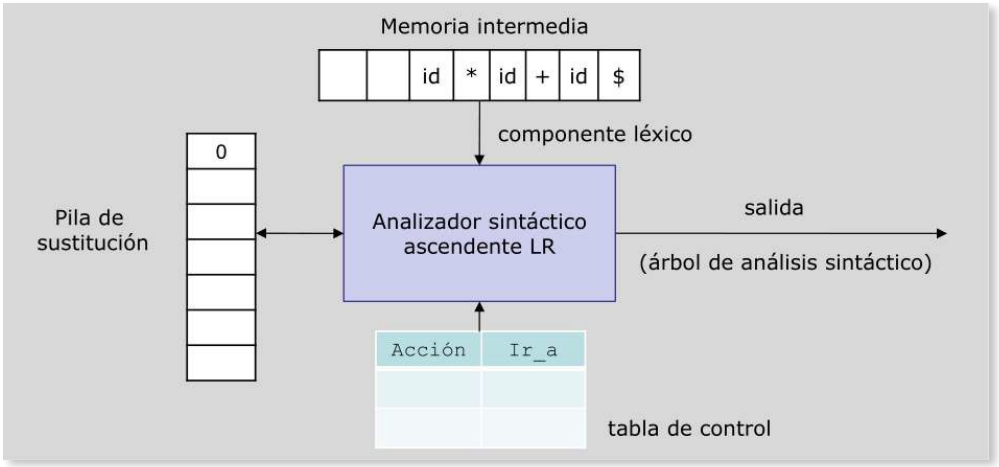
Procedimiento:

1. Construir la colección canónica $LR(0)=\{I_0, I_1, \dots, I_n\}+$
2. Para los elementos de I_j de la forma $A \rightarrow \alpha \bullet a \beta$, donde `a` es un terminal, e $Ir\_a(I_j, a) = I_k$, entonces `acción[j,a]` = **desplazar** la entrada e **ir al estado** correspondiente a I_k.
3. Para los elementos de I_j de la forma $A \rightarrow \alpha \bullet$, entonces `acción[j,s]` = **reducir** $A \rightarrow \alpha$, para cualquier $s \in$ `SIGUIENTES(A)`.
4. Si el elemento $S' \rightarrow S \bullet \$$ está en I_j, entonces `acción[j,$]` = **aceptar**.
5. Las entradas **vacías** de la tabla de acciones representan estados de **error**.

3.4.4. PROCEDIMIENTO DE ANÁLISIS

Algoritmo de análisis:

1. Inicializar la pila con el estado 0.
2. Para cada estado de la pila i y símbolo de la entrada a consultamos la celda acción[i, a].
    a. Si `acción[i,a]` = `dn`, entonces **desplazamos** la entrada, vamos al estado `n`, y lo introducimos en la cima de la pila.
    b. Si `acción[i,a]` = `rn`, entonces **reducimos** mediante la regla `n` de la forma $A \rightarrow \beta$; para ello, eliminamos de la pila tantos símbolos como longitud tenga β, y vamos al estado `Ir_a(h,A)`, donde `h` es el estado que queda en la cima de la pila.
    c. Si `acción[i,a]` = **aceptar**, completamos el análisis.
    d. Si `acción[i,a]` = ∅, estamos ante un **error** e invocamos la gestión de errores.



3.4.5. CONFLICTOS EN LAS TABLAS `SLR(1)`

Una **gramática es `SLR(1)`** si en la tabla de análisis aparece como máximo una entrada en cada celda. Cuando no es así pueden señalarse dos tipos de conflictos:

1. **Desplazamiento–reducción.** En la misma celda aparece una acción de desplazar y otra de reducir. Para poder continuar el análisis se elige una opción por defecto, normalmente desplazar.
2. **Reducción–reducción.** En un estado de la tabla, la misma entrada se puede reducir por dos reglas distintas. Se debe modificar la gramática.

3.4.6. GESTIÓN DE ERRORES

Cuando se produce un error, el estado de la pila representa el **contexto a la izquierda** del error, y el resto de la cadena de entrada, el **contexto a la derecha**. La recuperación del error se lleva a cabo modificando la pila y/o cadena de entrada, hasta que el estado y la cadena le permitan al analizador continuar el análisis.

Hay dos métodos de gestión de errores:

1. Los **métodos heurísticos** suponen la programación de rutinas específicas para cada celda en la que se produce un error.
2. El **método de análisis de transiciones** propone explorar la pila hacia abajo hasta encontrar un estado `d` con `Ir_a` para un no terminal `A`. Después se descartan cero o más símbolos en la entrada hasta encontrar uno que pueda seguir a `A` de acuerdo con la gramática. Esto permite meter el estado `Ir_a(s,A)` en la pila y continuar el análisis.

4. ANÁLISIS SEMÁNTICO

4.1. ANÁLISIS SEMÁNTICO

Cada símbolo de la gramática adquiere un determinado significado según el contexto de aparición en el código fuente. Para que los símbolos tengan significado, es necesario aumentar la gramática y añadir un conjunto de **atributos** a los símbolos y unas **acciones semánticas** a las reglas semánticas. Las especificaciones de la semántica se pueden realizar mediante:

- *Lenguajes especiales de especificación*
- *Gramáticas de atributos.*

4.1.1. GRAMÁTICAS DE ATRIBUTOS

Una **gramática de atributos** es una gramática independiente del contexto a la que se añade un sistema de atributos. Un sistema de atributos está formado por:

1. Un conjunto de **atributos semánticos** asociados a cada símbolo de la gramática.
2. Un conjunto de **acciones semánticas**, distribuidas a lo largo de las reglas de sustitución.

Un **atributo** de un símbolo `X` es una variable que representa una característica del símbolo y que puede tomar un valor de entre un conjunto, finito o no, de valores posibles. Se denota como `X.a`.

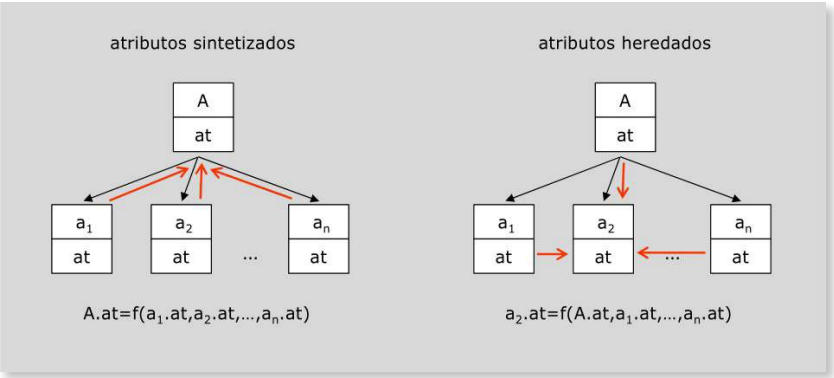
Una **acción semántica** es un algoritmo asociado a una regla de la gramática, cuyo objetivo es calcular el valor de alguno de los atributos de los símbolos de dicha regla.

4.1.2. PROPAGACIÓN DE ATRIBUTOS

Llamamos **propagación de atributos** al cálculo del valor de los atributos en función del valor de otros.

La propagación se realiza siguiendo el árbol de derivación que devuelve el análisis sintáctico. Según los símbolos que intervienen en la acción semántica, para calcular el valor del atributo de un símbolo, hay dos tipos de atributos:

1. **Atributos sintetizados.** Atributos asociados a no terminales de la parte izquierda de una regla de derivación. Se calculan a partir de los nodos hijos del subárbol en el que aparecen.
2. **Atributos heredados.** Atributos asociados a símbolos de la parte derecha de una sustitución. Se calculan a partir de los atributos del nodo padre y de los nodos hermanos dentro del mismo subárbol.



4.1.3. GRAFO DE DEPENDENCIAS

La propagación se ha de realizar siguiendo el orden implícito en la dependencia que existe entre los atributos. Este orden se representa mediante un **grafo de dependencias acíclico y dirigido**, para cuya construcción se representan las dependencias entre atributos de la forma $b = f(c_1, c_2, \ldots, c_n)$. El grafo tendrá un nodo para cada atributo de cada regla semántica, y un arco de c_i a b si b depende de c_i. Así:

1. Se evalúan los nodos a los que no llega ningún arco y se marcan como calculados.
2. Se evalúan los nodos en los que todos los arcos proceden de nodos marcados, y se marcan como calculados.
3. Continuamos hasta que no queden más nodos por evaluar.

4.1.4. VERIFICACIÓN DE TIPOS

La **verificación de tipos** asegura que el tipo de cada una de las construcciones del código fuente coincide con lo previsto en el diseño de la gramática.

Un **sistema de tipos** es una serie de reglas que permiten asignar tipos a las distintas construcciones del código fuente. Estas reglas se incluyen en el árbol de análisis sintáctico.

Un lenguaje es **fuertemente tipificado** si su compilador puede garantizar que los programas compilados se ejecutarán sin errores de tipo.

4.2. GENERACIÓN DE CÓDIGO INTERMEDIO

Una **representación intermedia** es una estructura de datos que representa al código fuente durante el proceso de traducción a código objeto. El **árbol de análisis sintáctico** es una representación intermedia, pero se parece poco al código ejecutable. Se propone una representación intermedia resultado de una linealización del árbol de análisis semántico, llamada **código de tres direcciones**, que se desarrolla para algún tipo de máquina virtual.

4.2.1. CÓDIGO DE TRES DIRECCIONES

El **código en tres direcciones** se define como una secuencia de instrucciones de la forma `x = y op z`, donde op representa a cualquier `operador` y `x`, `y`, `z` representan símbolos definidos por el programador, variables temporales generadas por el compilador, constantes, etc. Esta representación exige la descomposición y modificación de sentencias complejas y sentencias de flujo de control para dar lugar a sentencias en tres direcciones. Este código es una representación linealizada de izquierda a derecha del árbol de análisis sintáctico.

Para poder representar las construcciones de un lenguaje de alto nivel, es necesario introducir en el código intermedio:

1. **Instrucciones de asignación** con operador **binario** (`x := y op z`)
2. **Instrucciones de asignación** con operador **unario** (`x := op y`)
3. **Instrucciones de copia** (`x := y`)
4. **Salto incondicional** (`goto E`), donde `E` indica la siguiente instrucción a ejecutar
5. **Salto condicional** (`if x oprel y goto E`), donde se ejecuta la instrucción `E` o la siguiente en la secuencia de instrucciones dependiendo del resultado.
6. **Llamadas a procedimientos**
7. **Asignaciones con índices** (`x := y[i]` o `x[i] := y`)
8. **Asignaciones de direcciones y punteros** (`x := &y, x := *y` o `*x := y`)

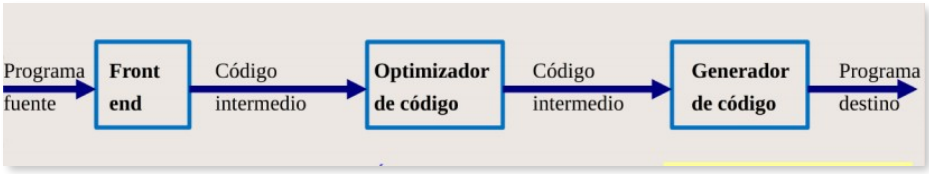
4.2.2. IMPLEMENTACIONES

El código intermedio suele almacenarse como una **lista enlazada de registros**, donde cada registro representa una instrucción. Existen diferentes implementaciones:

1. **Cuádruples.** Un **cuádruple** es una estructura de tipo registro con cuatro campos (`op, result, arg1` y `arg2`)
2. **Triples**. Un **triple** es una simplificación en la que cada instrucción representa a una variable temporal, usando sólo tres campos (`op, arg1` y `arg2`).

5. GENERACIÓN Y OPTIMIZACIÓN DE CÓDIGO (VER AHO)

A partir de la tabla de símbolos y la representación intermedia, puede obtenerse un programa destino semánticamente correcto y óptimo, añadiendo una fase de optimización.



En la fase de generación de código se debe:

1. Seleccionar las instrucciones
2. Repartir y asignar los registros
3. Ordenas las instrucciones

El criterio básico debe de ser la generación de un código correcto, mientras que el criterio secundario será la generación de un código de calidad.

El generador de código debe recibir:

1. Representación intermedia
2. Detección de errores sintácticos y semánticos
3. Comprobación de tipo de datos
4. Operadores de conversión de tipos

5.1. TAREAS PRINCIPALES

5.1.1. SELECCIÓN DE INSTRUCCIONES

El generador de código debe asignar el programa de representación intermedia a una secuencia de código que la máquina de destino pueda ejecutar. La complejidad de realizar esta asignación se determina mediante diversos factores:

1. Nivel de la representación intermedia
2. Conjunto de instrucciones
3. Calidad deseada

Para cada tipo de instrucción de tres direcciones, podemos diseñar un esqueleto de código que defina el código destino a generar para esa construcción. Por ejemplo, toda instrucción de tres direcciones de la forma `x = y + z`, en donde `x`, `y` y `z` se asignan en forma estática, puede traducirse en la siguiente secuencia de código:

```
LD R0, y        // R0 = y (carga y en el registro R0)
ADD R0, R0, z   // R0 = R0 + z (suma z a R0)
ST x, R0        // x = R0 (almacena R0 en x)
```

```
a= b + c
d = a + e
```

```
LD R0, b                LD R0, a
ADD R0, R0, c           ADD R0, R0, e
ST a, R0                ST d, R0
```

A menudo, esta estrategia produce operaciones de carga y almacenamiento redundantes.

5.1.2. ASIGNACIÓN DE REGISTROS

Los **registros** son la unidad computacional más veloz en la máquina destino pero, por lo general, no tenemos suficientes como para contener todos los valores. Los valores que no se guardan en registros deben residir en la memoria. Las instrucciones que involucran operandos de registros son invariablemente más cortas y rápidas que las que involucran el uso de operandos en la memoria, por lo que la utilización eficiente de los registros es muy importante.

El uso de los registros se divide en dos subproblemas:

4. **Repartición de registros**, durante la cual seleccionamos el conjunto de variables que residirán en los registros, en cada punto del programa.
5. **Asignación de registros**, durante la cual elegimos el registro específico en el que residirá una variable.

Es difícil encontrar una asignación óptima de registros a variables, incluso en las máquinas con un solo registro. En sentido matemático, el problema es NP-completo. El problema se complica aún más debido a que el hardware y el sistema operativo de la máquina destino pueden influir.

5.1.3. ORDEN DE EVALUACIÓN

El orden en el que se realizan los cálculos puede afectar la eficiencia del código destino. Ciertos órdenes de los cálculos requieren menos registros que otros para contener resultados inmediatos. Sin embargo, el proceso de elegir el mejor orden en el caso general es un problema NP-completo difícil. Al principio, debemos evitar el problema mediante la generación de código para las instrucciones de tres direcciones, en el orden en el que el generador de código intermedio las produjo.

5.2. EL LENGUAJE DESTINO

A menudo asociamos un costo con la compilación y la ejecución de un programa. Dependiendo del aspecto del programa que nos interese optimizar, algunas medidas de costos comunes son la longitud del tiempo de compilación y el tamaño, el tiempo de ejecución y el consumo de energía del programa de destino.

La determinación del costo actual de compilar y ejecutar un programa es un problema complejo. Buscar un programa destino óptimo para un programa fuente dado es un problema indecidible, y muchos de los subproblemas involucrados son $\texttt{NP} - \texttt{hard}$. Como hemos indicado antes, en la generación de código debemos a menudo contentarnos con las técnicas de heurística que producen buenos programas destino, pero que no necesariamente son óptimos.

El costo de una instrucción corresponde a la longitud en palabras de la instrucción. Los modos de direccionamiento que implican el uso de registros tienen un costo adicional de cero, mientras que los que implican el uso de una ubicación de memoria o constante tienen un costo adicional de uno, ya que dichos operandos tienen que almacenarse en las palabras que van después de la instrucción.

5.3. DIRECCIONES EN EL CÓDIGO DESTINO

Cada programa se ejecuta en su propio espacio de direcciones lógicas, que se particiona en cuatro áreas de código y cuatro de datos:

1. **Área de Código**
2. **Área de datos Estática**
3. **Área Montículo**
4. **Área Pila**

5.3.1. ASIGNACIÓN ESTÁTICA DE LLAMADAS A RUTINAS

En la **asignación estática de llamadas a rutinas** existe un registro estático para los argumentos de funciones y otro para los valores de retorno. Por este motivo, no puede haber llamadas recursivas, ya que los argumentos se sobreescribirían.

5.3.2. ASIGNACIÓN EN STACK DE LLAMADAS A RUTINAS

La **asignación estática** se puede convertir en **asignación de pila** mediante el uso de direcciones relativas para el almacenamiento en los registros de activación. Sin embargo, en la asignación de pila la posición de un registro de activación para un procedimiento no se conoce sino hasta el tiempo de ejecución. Por lo general, esta posición se almacena en un registro, por lo que se puede acceder a las palabras en el registro de activación como desplazamientos a partir del valor en este registro. El modo de direccionamiento indexado para nuestra máquina destino es conveniente para este fin.

5.4. BLOQUES BÁSICOS Y GRAFOS DE FLUJO

Un **grafo de flujo** es una representación gráfica del código intermedio, cuyos nodos son bloques básicos y sus flechas indican el orden de ejecución de los bloques. Este método resulta útil para la caracterización del código y la aplicación de técnicas en bloques y entre bloques.

Un **bloque básico** es una secuencia máxima de instrucciones consecutivas de tres direcciones, con las siguientes propiedades:

1. El flujo de control sólo puede entrar en el bloque básico a través de la primera instrucción del bloque. No hay saltos hacia la parte media del bloque.
2. El control saldrá sin detenerse o bifurcarse, excepto tal vez en la última instrucción del bloque. Algoritmo para la obtención de los bloques básicos:

Determinamos las instrucciones **líderes** del código intermedio, es decir, las primeras instrucciones de algún bloque básico. La instrucción que va justo después del programa intermedio no se incluye como líder. Las reglas para buscar líderes son:

1. La primera instrucción de tres direcciones en el código intermedio es líder.
2. Cualquier instrucción que sea el destino de un salto condicional o incondicional es líder.
3. Cualquier instrucción que siga justo después de un salto condicional o incondicional es líder.

Para cada instrucción líder, su bloque básico consiste en sí misma y en todas las instrucciones hasta la siguiente instrucción líder o el final del programa intermedio.

5.4.1. INFORMACIÓN DE SIGUIENTE USO

Es esencial saber cuándo se usará el valor de una variable para generar buen código. Si el valor de una variable que se encuentra en un registro nunca se utilizará más adelante, entonces ese registro puede asignarse a otra variable. Decimos que una variable está **viva** en una instrucción si posteriormente va a ser utilizada por otra.

Algoritmo para determinar la información de siguiente uso:

> **ENTRADA:** Un bloque básico `B` de instrucciones de tres direcciones. Suponemos que, al principio, la tabla de símbolos muestra que todas las variables no temporales en `B` están vivas al salir.
>
> **SALIDA**: En cada instrucción `i`: `x = y + z` en `B`, adjuntamos a `i` la información sobre la vida y el siguiente uso de `x, y, z`, donde + representa a cualquier operador.
>
> **MÉTODO:** Empezamos en la última instrucción en `B`, y exploramos en forma invertida hasta el inicio de `B`. En cada instrucción `i`: `x = y + z` en `B`, hacemos lo siguiente:
>
> 1. Adjuntar a la instrucción i la información que se encuentra en ese momento en la tabla de símbolos, en relación con el siguiente uso y la vida de `x, y, z`.
> 2. En la tabla de símbolos, establecer `x` a “no viva” y “sin siguiente uso”.
> 3. En la tabla de símbolos, establecer `y,z` a “viva” y los siguientes usos de `y,z` a `i`.

5.4.2. **GRAFOS DE FLUJO**

Una vez que un programa de código intermedio se particiona en bloques básicos, representamos el flujo de control entre ellos mediante un grafo de flujo. Los nodos del grafo de flujo son los nodos básicos. Hay una flecha del bloque `B` al bloque `C` sí, y sólo si es posible que la primera instrucción en el bloque `C` vaya justo después de la última instrucción en el bloque `B`. Hay dos formas en las que podría justificarse dicha flecha:

1. Hay un salto condicional o incondicional desde el final de `B` hasta el inicio de `C`.
2. `C` sigue justo después de `B` en el orden original de las instrucciones de tres direcciones, y `B` no termina en un salto incondicional.

A menudo agregamos dos nodos, conocidos como entrada y salida, los cuales no corresponden a instrucciones intermedias ejecutables. Hay una flecha que va desde la entrada hasta el primer nodo ejecutable del grafo de flujo; es decir, al bloque básico que surge de la primera instrucción del código intermedio. Hay una flecha que va a la salida desde cualquier bloque básico que contenga una instrucción, que pudiera ser la última instrucción ejecutada del programa. Pueden realizarse cambios considerables en los bloques, o aplicar técnicas de optimización entre bloques básicos.

5.4.3. **CICLOS**

Casi cualquier programa invierte la mayor parte de su tiempo en ejecutar **ciclos**, por lo que es importante generar un buen código para ellos. Muchas transformaciones dependen de la identificación de los ciclos en el grafo de flujo. Decimos que un conjunto de nodos L en un grafo es un ciclo si:

1. Hay un nodo en `L` llamado **entrada al ciclo** con la propiedad de que ningún otro nodo en `L` tiene un predecesor fuera de `L`. Cada ruta desde la entrada del grafo de flujo completo, hacia cualquier nodo en `L`, pasa a través de la entrada al ciclo.
2. Cada nodo en `L` tiene una ruta que no está vacía, completamente dentro de `L`, que va hacia la entrada de `L`.

5.5. OPTIMIZACIÓN DE LOS BLOQUES BÁSICOS

Muchas técnicas para la optimización local empiezan por transformar un bloque en un grafo dirigido acíclico. Se construye de la siguiente manera:

1. Hay un nodo en el grafo para cada uno de los valores iniciales de las variables que aparecen en el bloque básico.
2. Hay un nodo `N` asociado con cada instrucción s dentro del bloque. Los hijos de `N` son aquellos nodos que corresponden a las instrucciones que son las últimas definiciones, anteriores a `s`, de los operandos utilizados por `s`.
3. El nodo `N` se etiqueta mediante el operador que se aplica en `s`, y también a `N` se adjunta la lista de variables para las cuales es la última definición dentro del bloque.
4. Ciertos nodos se designan como nodos de salida, si sus variables están vivas al salir del bloque.

La representación en grafo de un bloque básico permite realizar transformaciones para mejorar el código dentro del bloque:

1. **Eliminar subexpresiones locales comunes**, instrucciones que calculan un valor que ya ha sido previamente calculado.
2. **Eliminar código muerto,** instrucciones que calculan un valor que nunca se utiliza.
3. **Reordenar las instrucciones** que no dependen unas de otras.
4. **Aplicar leyes algebraicas** para reordenar los operandos de las instrucciones de tres direcciones.
    a. **Aplicación de leyes aritméticas**
    b. **Reducción por fuerza local**. Sustituir un operando costoso por otro más económico

c. **Plegado de constantes**. Se evalúan las expresiones constantes en tiempo de compilación y se sustituyen por sus valores.

d. **Conmutatividad y asociatividad**. Debe tenerse cuidado con ellas ya que la aritmética de computadora no siempre obedece a las identidades algebraicas de las matemáticas.

5.5.1. REPRESENTACIÓN DE REFERENCIAS A ARRAYS

Las instrucciones donde aparecen arrays con índices no reciben el mismo tratamiento que cualquier otro operador. Si consideramos a `a[i]` como una operación en la que se involucran `a` e `i`, similar a `a+i`, entonces podría parecer que los dos usos de `a[i]` son una subexpresión común.

Consideremos la siguiente secuencia:

```
x = a[i]
a[j] = y
z = a[i]
```

En este caso, podríamos vernos tentados a "optimizar" mediante la sustitución de la tercera instrucción `z = a[i]` por una más simple, `z = x`. Sin embargo, como `j` podría ser igual a `i`, la instrucción de en medio puede de hecho modificar el valor de `a[i]`; por ende, no es legal realizar esta modificación.

La manera apropiada de representar los accesos a arrays en un GDA es la siguiente:

1. Una asignación de un arreglo como `x = a[i]` se representa mediante la creación de un nodo con el operador `=[ ]` y dos hijos, los cuales representan el valor inicial del arreglo, en este caso a_0, y el índice i. La variable x se convierte en una etiqueta de este nuevo nodo.
2. Una asignación a un arreglo, como `a[j] = y`, se representa mediante un nuevo nodo con el operador `[ ]=` y tres hijos que representan a a_0`,j,y`. No hay una variable para etiquetar este nodo. La diferencia es que la creación de este nodo elimina a todos los nodos actuales construidos, cuyo valor depende de a_0. Un nodo que se ha eliminado no puede recibir más etiquetas; es decir, no puede convertirse en una expresión común.

5.5.2. ASIGNACIONES DE PUNTEROS Y LLAMADAS A PROCEDIMIENTOS

Cuando realizamos una asignación indirecta a través de un puntero, no sabemos a dónde apunta en realidad. En realidad `x = *p` es un uso de cualquier variable, y `*q = y` es una posible asignación a cualquier variable. Como consecuencia, el operador `=*` debe aceptar todos los nodos que se encuentran asociados con identificadores como argumentos, lo cual es relevante para la eliminación de código muerto. Lo más importante es que el operador `*=` elimina a todos los demás nodos construidos hasta el momento en el GDA.

El caso particular:

```
p = &x
*p = y
```

Implica que sólo la variable `x` recibe el valor de `y`, por lo que no se necesita eliminar ningún nodo, a excepción del nodo al cual se adjunta `x`.

5.5.3. REENSAMBLADO DE LOS BLOQUES BÁSICOS A PARTIR DEL GDA

Después de haber realizado todas las optimizaciones posibles sobre el GDA, podemos reconstruir el código de tres direcciones para el bloque básico a partir del cual se ha creado el GDA.

Para cada nodo que tenga una o más variables adjuntas, construimos una instrucción de tres direcciones que calcula el valor de una de estas variables. Preferimos calcular el resultado en una variable que esté viva al salir del bloque.

No obstante, si no tenemos información sobre las variables globales vivas en la cual podamos trabajar,

debemos suponer que todas las variables del programa (pero no las temporales que genera el compilador para procesar expresiones) están vivas al salir del bloque.

Si el nodo tiene más de una variable viva adjunta, entonces tenemos que introducir instrucciones de copia para dar el valor correcto a cada una de esas variables. Algunas veces la eliminación global puede eliminar esas copias, si podemos arreglárnoslas para usar una de dos variables en vez de la otra.

5.6. UN GENERADOR DE CÓDIGO SIMPLE

Una de las cuestiones principales durante la generación de código es decidir cómo usar los registros para poder sacarles el máximo provecho.

Usos principales de los registros**:**

- En la mayoría de las arquitecturas, algunos o todos los operandos de una operación deben estar en registros para poder llevarla a cabo.
- Los registros son excelentes variables temporales que sirven para almacenar el resultado de una subexpresión mientras se evalúa una expresión más grande.
- Los registros se utilizan para guardar valores que se calculan en un bloque básico y se utilizan en otros bloques.
- A menudo, los registros se utilizan para ayudar con la administración del almacenamiento en tiempo de ejecución, por ejemplo, para administrar la pila.

5.6.1. DESCRIPTORES DE REGISTROS Y DIRECCIONES

El algoritmo de generación de código analiza una instrucción de tres direcciones a la vez y decide qué operaciones de carga son necesarias para meter los operandos requeridos en los registros. Para poder realizar las decisiones necesarias, requerimos una estructura de datos que nos indique qué variables del programa tienen actualmente su valor en un registro, y en qué registro o registros están, si es así. También debemos saber si la ubicación de memoria para una variable dada tiene actualmente el valor apropiado para esa variable, ya que tal vez se haya calculado un nuevo valor para esa variable en un registro y todavía no se almacena. La estructura de datos deseada tiene los siguientes descriptores:

1. Para cada registro disponible, un descriptor de registro lleva la cuenta de los nombres de las variables cuyo valor actual se encuentra en ese registro.
2. Para cada variable del programa, se almacenan las posiciones donde puede encontrarse.

5.6.2. ALGORITMO DE GENERACIÓN DE CÓDIGO

Una parte esencial del algoritmo es una función llamada `obtenReg(I)`, que selecciona registros para cada ubicación de memoria asociada con la instrucción de tres direcciones `I`. Al final del bloque básico, genera almacenamientos para las variables vivas.

5.6.3. FUNCIÓN `obtenReg`

Partiendo de la instrucción genérica `x = y + z`, se selecciona un registro para `y` y otro para `z`.

Selección de un **registro** para **`y`**:

1. Si `y` se encuentra en un registro, se elige un registro que ya contenga a `y`. No se genera ninguna instrucción de máquina para cargar `y`.
2. Si `y` no está en un registro, pero hay un registro vacío, se elige dicho registro.
3. Si `y` no se encuentra en un registro, y no hay registros vacíos, hay que elegir un registro permitido y asegurarse de que sea seguro volver a utilizarlo. Supongamos que `R` es un registro candidato, y que `v` es una de las variables que el descriptor de registro para `R` dice que está en `R`. Tenemos que estar seguros de que el valor de `v` no se necesite realmente, o que haya alguna otra parte a la que podamos ir para obtener el valor de `R`.

Las posibilidades son:

a. Si el descriptor de dirección para `v` dice que está en alguna otra parte además de `R`, no hay problema.
b. Si `v` es `x`, el valor calculador por la instrucción `Y`, y `x` no es tampoco uno de los otros operandos de la instrucción `I`, no hay problema.
c. Si `v` no se utiliza más adelante (no hay más usos de `v` después de la instrucción `I`, y `v` está viva al salir del bloque), no hay problema.
d. Si no se cumple ninguno de los casos anteriores, debemos generar una instrucción de almacenamiento para colocar una copia de `v` en su propia ubicación de memoria.

Selección de un **registro** para **`x`**: (similar al caso de y, sólo se mencionan las diferencias)

1. Un registro que contiene a `x` es siempre una opción aceptable, incluso si `x` es `y` o `z`, ya que nuestras instrucciones de máquina permiten que dos registros sean iguales en una instrucción.
2. Si `y` no se utiliza después de la instrucción `I`, y el registro que contiene a `y` sólo lo contiene a él después de cargarse, si es necesario, puede ser utilizado como registro para `x`.

En caso de que `I` sea una instrucción de copia `x = y`, elegimos un registro para `y` de la forma explicada anteriormente, y luego elegimos $R_x = R_y$.

5.7. Generación de código a partir de árboles de expresión etiquetados

En nuestro modelo de máquina, en donde todos los operandos deben estar en registros, y los registros pueden utilizarse tanto por un operando como por el resultado de una operación, la etiqueta de un nodo es la menor cantidad de registros con los que se puede evaluar la expresión, sin utilizar almacenamientos de resultados temporales.

Como en este modelo estamos obligados a cargar cada operando y a calcular el resultado correspondiente a cada nodo interior, lo único que puede hacer que el código generado sea inferior al código óptimo es si hay almacenamientos innecesarios de valores temporales.

El argumento para esta declaración está incrustado en el siguiente algoritmo para generar código sin almacenamientos de valores temporales, utilizando un número de registros igual a la etiqueta de la raíz.

Algoritmo:

ENTRADA: Un árbol etiquetado con cada operando que aparece una vez.

SALIDA: Una secuencia óptima de instrucciones de máquina para evaluar la raíz y colocarla en un registro.

MÉTODO: Los siguientes pasos se aplican, empezando en la raíz del árbol. Si el algoritmo se aplica a un nodo con la etiqueta `k`, entonces sólo se utilizarán `k` registros. Pero hay una "base" `b>=1` para los registros utilizados, de manera que los registros actuales utilizados son $R_b, R_{b+1}, \ldots, R_{b+k-1}$.
El resultado siempre aparece en R_{b+k-1}.

1. Para generar código máquina para un nodo interior con la etiqueta `k` y dos hijos con etiquetas iguales (que deben ser $k-1$), haga lo siguiente:
   a. Genere código en forma recursiva para el hijo derecho, usando la base $b+1$. El resultado del hijo derecho aparece en el registro R_{b+k}.
   b. Genere código en forma recursiva para el hijo izquierdo, usando la base `b`; el resultado aparece en R_{b+k-1}.
   c. Genere la instrucción OP $R_{b+k}, R_{b+k-1}, R_{b+k}$, en donde OP es la operación apropiada para el nodo interior en cuestión.
2. Suponga que tenemos un nodo interior con la etiqueta `k` e hijos con etiquetas desiguales. Entonces uno de los hijos, al que llamaremos el hijo "grande", tiene la etiqueta `k`, y el otro

hijo, el hijo “pequeño”, tiene cierta etiqueta $m < k$. Haga lo siguiente para generar código para este nodo interior, usando la base b:

a. Genere código en forma recursiva para el hijo grande, usando la base b; el resultado aparece en el registro R_{b+k-1}.

b. Genere código en forma recursiva para el hijo pequeño, usando la base b; el resultado aparece en el registro R_{b+m-1}. Observe que como $m < k$ no se utiliza R_{b+k-1} ni cualquier otro registro con numeración más alta.

c. Genere la instrucción

OP R_{b+k-1}, R_{b+m-1}, R_{b+k-1}

o la instrucción

OP R_{b+k-1}, R_{b+k-1}, R_{b+m-1},

dependiendo de si el hijo grande es el derecho o el izquierdo, respectivamente.

3. Para una hoja que represente al operando x, si la base es b genere la instrucción LD R_b,x.