FACULTAD DE INFORMÁTICA
UNIVERSIDAD POLITÉCNICA DE MADRID

UNIVERSIDAD POLITÉCNICA DE MADRID
FACULTAD DE INFORMÁTICA

TRABAJO FIN DE CARRERA

INTERPRETE DE UN LABORATORIO VIRTUAL DE ADN

AUTOR: JAVIER GÓMEZ CONTRERAS
TUTOR: JUAN B. CASTELLANOS PEÑUELA

íNDICE

1 Prologo

Este trabajo fin de carrera ha sido realizado con gran esfuerzo, entusiasmo e ilusión, no solo por terminar la carrera y obtener el titulo de Ingeniero en Informática, sino porque este proyecto puede servir, y espero que así sea, para mejorar el desarrollo de protocolos realizados por ADN.

Con la propuesta del tutor y en colaboración con el cotutor llegamos a la idea de realizar un simulador de un laboratorio virtual de ADN. Este fue elegido por las posibilidades que nos ofrecía, como empezar a documentar protocolos, que aun funcionando en el simulador, en la realidad sería muy distinto, por las complejidades de las operaciones del ADN, otra sería por las posibilidades que ofrecen las biocomputadoras.

Además el tutor y cotutor me dieron libertad para elegir el lenguaje de programación, quería un lenguaje que me sirviera para el futuro, así que, elegí el lenguaje de Microsoft .NET c# debido a que es el mas adecuado para el desarrollo de la interfaz gráfica.

Siempre he pensado que todo lo que hacemos debe de ser de utilidad. Aunque es difícil de definir que es útil en esta sociedad, yo he creído que este proyecto podría ayudar a avanzar, pero lo que más me ha dado fuerzas, es que Dios haya querido que lo realizase, no se muy bien el motivo, pero eso es lo de menos. Lo importante es que esta hecho con sus mas y sus menos. Solo pido que le sea de mayor agrado al Sagrado Corazón de Jesús.

2 Agradecimientos

Quiero dar mis más sinceros agradecimientos a mi familia por haberme aguantado hasta hartarse, porque ya era hora de que terminase el proyecto. A mi novia por el ejemplo que me ha dado y me sigue dando, para poder ser una persona más buena, sencilla y un pelín más santa. A Juan Castellanos por su disponibilidad en todos los aspectos, en lo material por dejarme trabajar en su departamento, en su preocupación para espabilarnos en terminar la carrera, y por su servicialidad al prestarnos siempre lo que le hemos pedido. Al cotutor, Fernando Arroyo, por ese salero que el tiene, y por la paciencia que ha tenido para enseñarnos lo necesario para ir desarrollando el proyecto. A mis amigos, en especial las personas que forman o han formado el Grupo de Oración del Corazón de Jesús por sus intensas oraciones, y en especial al grupo de campamento.

3 Objetivos.

- Implementación de protocolos para un laboratorio virtual de ADN, con un sencillo manejo e interpretación del mismo.

4 Desarrollo

4.1 Teoría de grafos

Un grafo es una pareja **G = (V, A)**, donde **V** es un conjunto de puntos, llamados vértices o nodos, y **A** es un conjunto de pares de vértices, llamadas aristas o arcos. Para simplificar, la arista {a, b} se denota ab.

Ejemplo:

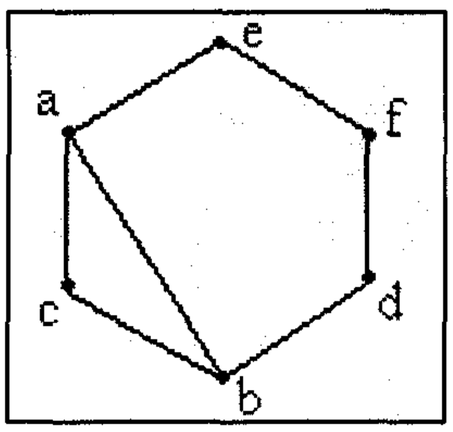


Figura 4.1.1. Grafo.

En la figura 4.1.1, V = {a, b, c, d, e, f }, y A = { ab, ac, ae, bc, bd, df, ef }.

En algunos casos es necesario imponer un sentido a las aristas, por ejemplo, si se quiere representar la red de las calles de una ciudad con sus inevitables direcciones únicas. Las aristas son entonces pares ordenados de vértices, con (a,b) ≠ (b,a), y se define así **grafos orientados**, como el siguiente:

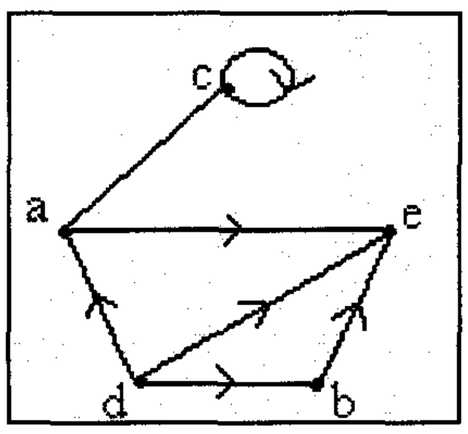


Figura 4.1.2. Grafo orientado.

En este grafo se ha autorizado una arista que tiene sus dos cabos idénticos: es un rizo (o bucle), y aparece también una arista sin flecha: significa que la arista se puede recorrer en cualquier sentido: es bidireccional, y corresponde a dos aristas orientadas.

Aquí V = { a, b, c, d, e }, y A = { **(a,c)**, (d,a), (a,e), (b,e), **(c,a)**,**(c,c)**, (d,b) }.

Del vértice *d* sólo salen vértices: es una fuente. Al vértice *e* sólo entran vértices: es un agujero, o pozo.

Un **ciclo** es un camino, es decir una sucesión de aristas adyacentes, donde no se recorre dos veces la misma arista, y donde se regresa al punto inicial. Un **ciclo hamiltoniano** tiene además que recorrer todos los vértices. Por ejemplo, en un museo grande (al estilo del Louvre), lo idóneo sería recorrer todas las salas una sola vez, esto es buscar un ciclo hamiltoniano en el grafo que representa el museo (los vértices son las salas, y las aristas los corredores o puertas entre ellas).

Se habla también de **camino hamiltoniano** si no se impone regresar al punto de partida, como en un museo con una única puerta de entrada. Por ejemplo, un caballo puede recorrer todas las casillas de un tablero de ajedrez sin pasar dos veces por la misma: es un camino hamiltoniano.

En la siguiente imagen se representa un grafo dodecaedro y un posible camino hamiltoniano. Este se representa de forma más gruesa.

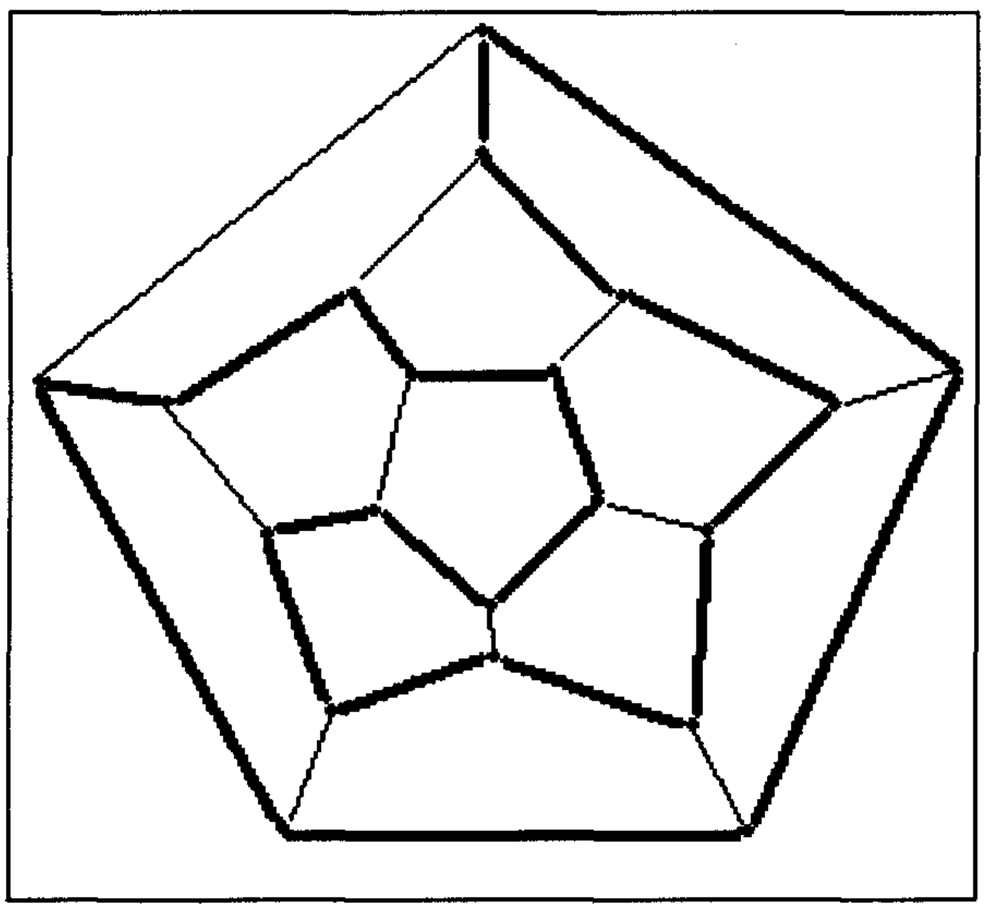


Figura 4.1.3. Grafo mostrando camino hamiltoniano.

No se conocen métodos generales para hallar un ciclo hamiltoniano en tiempo polinómico, siendo la búsqueda en bruto de todos los posibles caminos u otros métodos excesivamente costosos. Este problema está en el conjunto de los NP-Completo.

4.2 Gramáticas formales

4.2.1 Lenguaje formal

Conjunto de secuencias de símbolos de un alfabeto, que cumplen una propiedad.

4.2.2 Operaciones con lenguajes

- Unión.

$$L_1 \cup L_2 = \{x | x \in L_1 \vee x \in L_2 \}$$

- Intersección.

$$L_1 \cap L_2 = \{x | x \in L_1 \wedge x \in L_2 \}$$

- Concatenación.

$$L_1 L_2 = \{xz | x \in L_1 \wedge z \in L_2 \}$$

- Clausura.

$$L^* = \bigcup_{i=0}^{\infty} L^i$$

4.2.3 Gramática formal

Define construcciones de un lenguaje de forma recursiva. Para ello involucra cuatro identificadores:

G=(T,N,S,P)

-T. Conjunto de símbolos terminales que identifican a los elementos de la gramática.

Se representan mediante letras minúsculas del propio abecedario, símbolos de operador, símbolos de puntuación, dígitos del 0 al 9 o cadenas de caracteres que representan palabras claves.

-N. Conjunto de símbolos no terminales.

Se representan mediante nombres en minúscula o letras mayúsculas del abecedario.

-S. Axioma o símbolo principal del lenguaje.

S pertenece N

-P. Conjunto de producciones o reglas de derivación.

Por tanto, dada una gramática G = (T,N,S,P), se define el lenguaje L(G) como el conjunto de cadenas de símbolos terminales generados, partiendo del símbolo inicial S y empleando las reglas de producción P.

4.2.4 Clasificación de gramáticas

-Gramática tipo 0. Son gramáticas sin restricciones y sus producciones son de la forma:

P: $\alpha \rightarrow \beta$

con $\alpha \in (N \cup T)^*, \beta \in (N \cup T)^*$

Gramática tipo 1. Son gramáticas dependientes del contexto y sus producciones son de la forma:

P: $\alpha A \beta \rightarrow \alpha \gamma \beta$

con $A \in N, \alpha, \beta \in (N \cup T)^*, \gamma \in (N \cup T)^+$

Gramática tipo 2. Son gramáticas independientes del contexto y sus producciones son de la forma:

P: $A \rightarrow \alpha$

con $A \in N, \alpha \in (N \cup T)^*$

Gramática tipo 3. Son gramáticas regulares y sus producciones son de la forma:

P: $A \rightarrow aB$

$A \rightarrow a$

con $A, B \in N, a \in T$

4.3 Autómatas Finitos

4.3.1 Expresiones regulares

Las reglas que definen expresiones regulares para un alfabeto E dado son las siguientes:

1. λ es una expresión regular que representa la secuencia vacía.

2. Si a pertenece E es una expresión regular, entonces {a} es una expresión regular.

3. Sea r y s dos expresiones regulares que denotan al lenguaje L(r) y L(s). Se cumple:

-(r)|(s) es una expresión regular notada por L(s) unión L(s).

-(r)(s) es una expresión regular notada por L(r)L(s).

-(r)* es una expresión regular notada por L(r)*.

-(r) es una expresión regular notada por L(r).

A toda expresión regular le corresponde un autómata finito no determinista (AFND).

4.3.2 Autómata finito no determinista

Existe el modelo de construcción de Thopson para obtener el AFND correspondiente a una expresión regular dada. Este modelo, a partir de las siguientes expresiones regulares, asocia los siguientes AFND:

-Para la secuencia vacía λ. Figura 4.2.1.

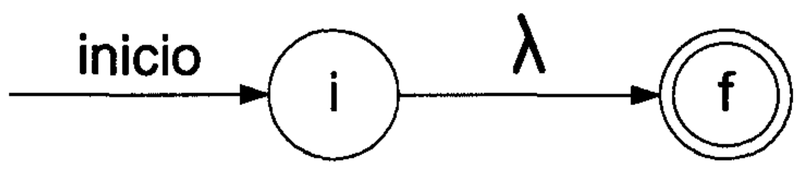


Figura 4.2.1.

-Para la secuencia a pertenece E. Figura 4.2.2.

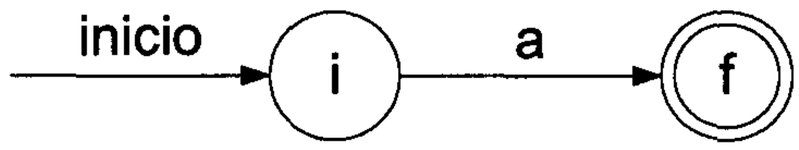


Figura 4.2.2.

Sean s y t expresiones regulares y N(s) y N(t) los AFNDs correspondientes, tenemos:

-Para s|t. Figura 4.2.3.

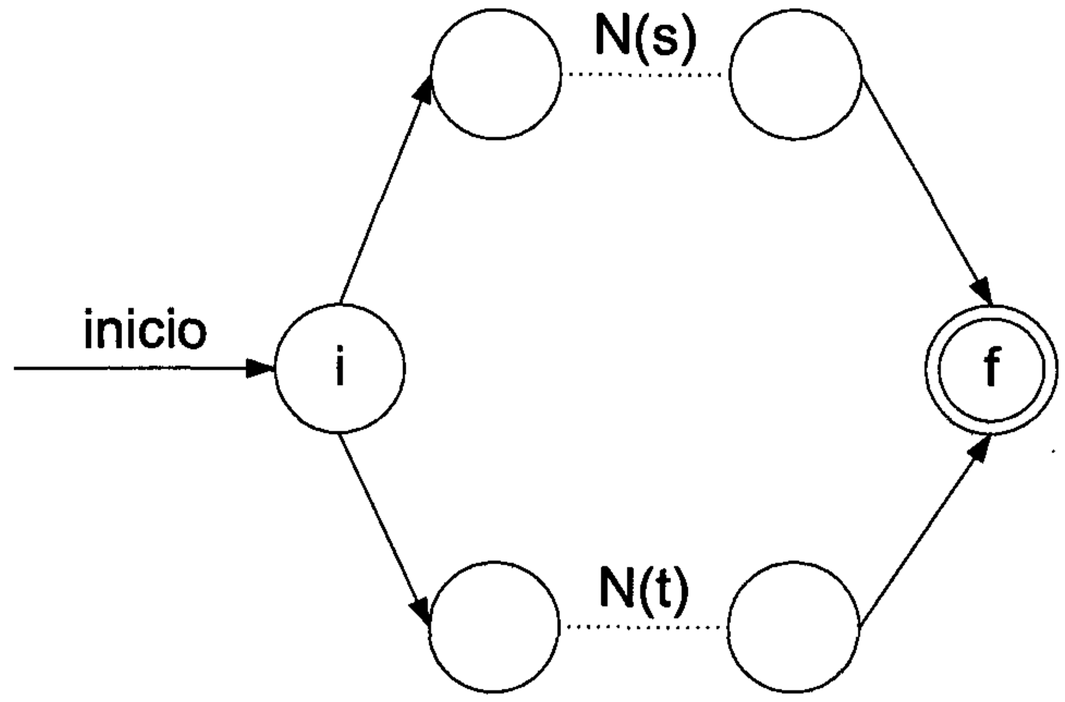


Figura 4.2.3.

-Para st. Figura 4.2.4.

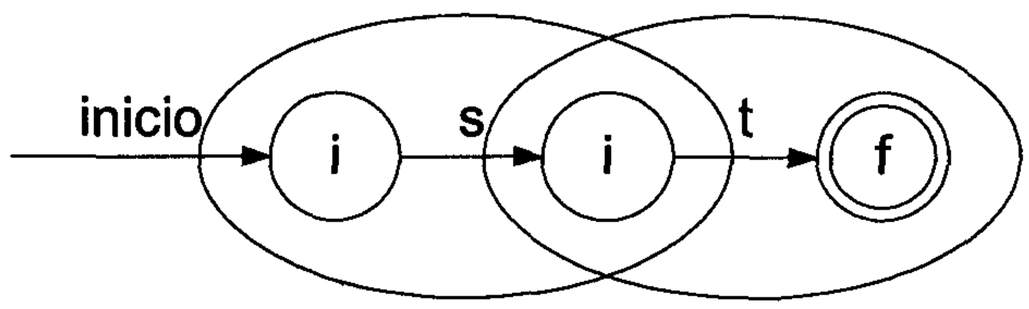


Figura 4.2.4.

-Para expresiones s* se construye el AFND N(s*). Figura 4.2.5.

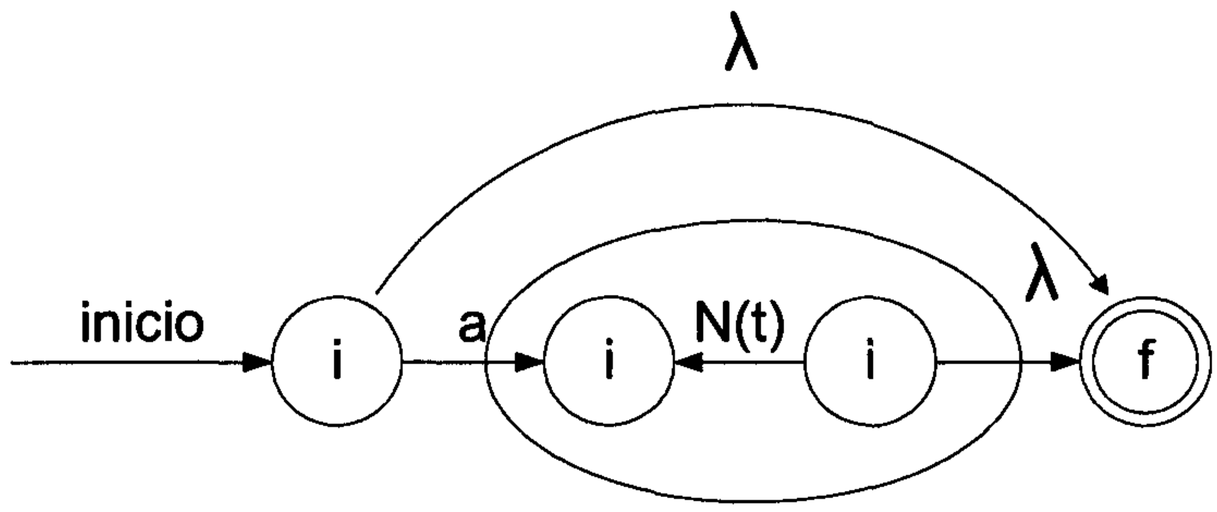


Figura 4.2.5.

4.3.3 Autómata finito determinista

Todo AFND es posible obtener a partir de él un AFD, que sea equivalente.

Se puede considerar AFD como un caso especial de un AFND, en el cual se dan las siguientes condiciones:

1. Ningún estado tiene λ-transición.

2. Para cada estado s y cada símbolo de entrada a, hay a lo sumo una arista etiquetada a que sale de s.

Para la conversión emplearemos tres funciones:

-λ-clausura(s). Conjunto de estados del AFND alcanzables desde el estado s únicamente con λ-transiciones.

- λ -clausura(T). Conjunto de estados del AFND alcanzables desde algún estado s en T únicamente con λ-transiciones (siendo T un conjunto de estados).

-Move(T,a). Conjunto de estados del AFND hacia los cuales hay una transición con el símbolo de entrada a desde algún estado s en T.

4.3.4 Reconocimiento de patrones

Sea P1|P2|...|Pn

los patrones que deberá reconocer, se le asociará a cada patrón un AFND

Pi-->N(Pi)

se añade una λ-transición y un estado inicial. Figura 4.2.6.

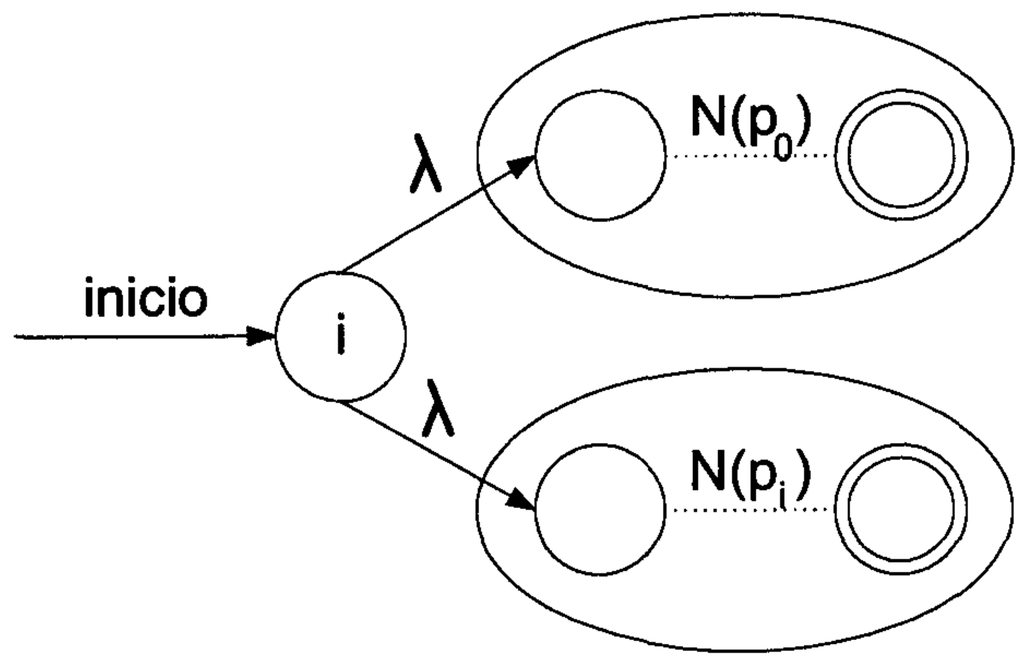


Figura 4.2.6.

4.4 Técnicas de laboratorio

4.4.1 Funciones de computación.

Un computador tiene dos capacidades básicas:

- Almacenamiento de la información.
- Procesamiento de la información.

Nos centraremos en el tema del procesamiento de la información de las hebras de ADN. Para ello necesitamos saber que técnicas de biología molecular podrían ser utilizadas para dicha computación y tener conocimiento de cómo se van a almacenar estos datos.

El almacenamiento de la información biológica se encuentra principalmente en el ADN, que se encuentra en todo organismo celular. La información que guarda el ADN esta compuesto por unidades biológicas llamadas nucleótidos, distinguibles entre si por su grupo químico o base: unida a ellos. Las cuatro bases que existen son:

- Adenina.
- Guanina.
- Citosina.
- Timina.

Desde ahora lo representaremos con las abreviaturas A, G, C, T, respectivamente.

Se consideran las siguientes operaciones básicas de ADN:

- Annealing.
- Amplificación.
- Corte.
- Destrucción.
- Detección.
- Extracción.
- Lectura.

- Marcado.
- Melting.
- Mezcla.
- Separación.
- Síntesis.
- Substitución.
- Unión.

4.4.1.1 Annealing

Enlaza dos cadenas de ADN simples y complementarias a través del enfriamiento de la solución en la que se encuentren. Esta operación recibe el nombre de "hibridación" si la operación de annealing se realiza in Vitro.

Funcionalidad: Enlaza todas las posibles cadenas simples donde puedan darse esa unión, incluso haciendo replicas de las cadenas simples o nucleótidos que se tenga. Devuelve cadenas de ADN.

Ejemplo: Para tres cadenas simples TTT, AAGG, C, devolvería las cadenas:

```
TTT        TTT C      TTT CC       TTT        TTTC       TTTCC
  AAGG       AAGG       AAGG         AAGG       AAGG       AAGG
```

4.4.1.2 Amplificación

Crea una copia de cadenas de ADN usando una técnica llamada PCR (Polymerase Chain Reaction). La reacción para replicar una cadena requiere de una cadena simple que servirá de guía, llamada "patrón", y un oligonucleótido corto llamado "primer" que se enlaza al patrón.

Se busca el patrón en la cadena a amplificar y se extiende el "primer" con los nucleótidos necesarios hasta completar la cadena original.

4.4.1.3 Corte

Realiza el corte de cadenas compuestas de ADN en sitios específicos utilizando enzimas de restricción. Una clase de enzimas, llamadas "endonucleasas de restricción" reconocen un patrón específico en la secuencia del ADN.

Una vez localizadas todas las posiciones donde se encuentra el patrón dentro de la secuencia del ADN se utiliza la enzima para cortar la cadena en esas localizaciones.

4.4.1.4 Destrucción

Permite destruir las cadenas marcadas utilizando exonucleasas, o cortando todas con una enzima de restricción y eliminándolas después con un gel electroforesis.

4.4.1.5 Detección

Detecta si el tubo de ensayo contiene alguna cadena de ADN.

4.4.1.6 Extracción

Selecciona cadenas que contienen un patrón dado dentro de la cadena de ADN.

4.4.1.7 Lectura

Obtiene la secuencia de nucleótidos del que se compone una cadena de ADN.

4.4.1.8 Marcado

Mediante "hibridación" secuencias complementarias se enlazan a las cadenas simples convirtiéndolas en cadenas dobles complementarias. La operación inversa es el "desmarcado" y consiste en desnaturalizar esas dobles cadenas complementarias desenlazando de nuevo las cadenas simples de la cadena de ADN.

4.4.1.9 Melting

Separa una cadena doble de ADN en sus cadenas de ADN simples y complementarias que la forman. Esto se consigue calentando la solución en la que se encuentra. La operación de "melting" in Vitro se conoce también como "desnaturalización".

4.4.1.10 Mezcla

Realiza la mezcla de dos tubos de ensayo en un tercero para conseguir la unión de los anteriores. La mezcla se puede conseguir rehidratando el contenido de los tubos (en el caso de que no estuvieran en solución) y entonces combinando sus fluidos juntos en un nuevo tubo.

4.4.1.11 Separación

Separa las cadenas por cierta longitud usando una técnica llamada "electroforesis"; las moléculas son situadas en la parte alta de un gel húmedo, al cual se le aplica un campo eléctrico, provocando que las moléculas se trasladen hacia la parte baja del gel.

Las moléculas más grandes avanzan con mayor lentitud a través del gel, con lo cual después de transcurrido un periodo determinado de tiempo, las moléculas se distribuyen en distintas bandas de acorde a su tamaño.

4.4.1.12 Síntesis

Creación de una molécula de ADN en estado sólido. Esta molécula es construida nucleótido a nucleótido sobre una partícula soporte en una serie de pasos consecutivos: el primer nucleótido (por ejemplo, A) es introducido en una solución específica en un recipiente de cristal, que se mantendrá fijado a la superficie inferior del recipiente. A continuación se vierte otra solución que contenga únicamente otro nucleótido (por ejemplo, C), el cual reacciona químicamente con el anterior nucleótido formando un elemento con 2 nucleótidos (AC). Tras eliminar el exceso de solución del nucleótido C, se repite el proceso para unir la cadena AC (cuyo extremo C puede seguir reaccionando) con otro nucleótido (por ejemplo, T) formando la cadena ACT.

Este proceso construye cadenas de ADN de cualquier longitud y de la secuencia de nucleótidos que se desee.

4.4.1.13 Substitución

Sustituye, inserta o elimina secuencias de ADN en una cadena de ADN. Se busca el patrón en la cadena a través de una enzima determinada y a continuación, mediante mutagénesis se realiza la operación pertinente.

4.4.1.14 Unión

Pega cadenas de ADN cuyos extremos sean complementarios (lo que se denominan "sticky end" compatibles) a través de ligasas de ADN, que son enzimas que realizan el trabajo contrario a las enzimas de corte.

4.5 Experimento L. M. Adleman

Resuelve una instancia concreta del problema del camino hamiltoniano, en su versión dirigida y con un par de nodos distinguidos a través de computación molecular.

Esto realizo L. M. Adleman, revolucionando el mundo de la computación molecular. Gracias a este experimento se abrió la posibilidad de poder tratar problemas NP-Completos, debido a que las operaciones con el ADN son operaciones NP-completas y este el primer experimento que logró solucionar un problema NP-Completo con estas operaciones.

Muchos científicos han empezado a dar soluciones a algoritmos bajo la computación molecular, aunque sea de forma teórica. Adleman consiguió resolverlo realmente en el laboratorio.

4.5.1 Trayectoria L.M. Adleman.

L.M. Adleman es un matemático que se doctoró en informática por la Universidad de California, Berkeley, en el año 1976. A finales de la década de los ochenta simultanea su labor docente e investigadora en el Instituto de Tecnología de Massachussets (MIT) con unas investigaciones sobre el virus VIH del sida. Los resultados parece ser que fueron bastantes modestos, hasta tal grado que se rechazaron para publicarse en revistas especializadas, lo que represento un duro traspiés para él. Sin embargo, no se desilusionó en su entusiasmo, en estos momentos y gracias al trabajo anterior toma contacto con N. Chelyapov que lo introduce en el mundo de la biología molecular.

Empezó a leer libros, y en uno de ellos, Biología molecular del gen, de J. Watson, le hizo ver, según sus palabras, que en la biología molecular había matemáticas, ya que una parte central de la misma estaba dedicada al estudio de la información codificada por las bases nucleótidas, así como al análisis de las transformaciones que experimenta dicha información en el interior de los organismos vivos.

También descubrió que la acción de la enzima polimerasa para reproducir una copia complementaria de una cadena simple de ADN tenía una gran similitud con el proceso mecánico que desarrolla una máquina de Turing. La polimerasa se va deslizando sobre una cadena simple de ADN (en una dirección fija), va leyendo cada una de las bases nucleótidas que encuentra y va escribiendo una adecuada copia (complementaria de la leída) formando otra cadena simple de ADN (enlazada con la anterior por el principio de complementariedad de Watson-Crick). Por otra parte, la cabeza de trabajo de una máquina de Turing se desplaza sobre la cinta (en distintas direcciones), lee el contenido de la celda que está analizando y, en su caso, escribe sobre ella.

L.M. Adleman trató de construir una máquina molecular que usara moléculas de ADN como sustrato computacional y que simulara exactamente el comportamiento de una máquina de Turing determinista. Su idea era encontrar unas enzimas (como la polimerasa) que proporcionaran una implementación molecular de las operaciones básicas de dicha máquina. Este intento no tuvo éxito (el equipo de E. Shapiro lo ha conseguido materializar en noviembre de 2001).

A pesar del fracaso, decidió atacar la resolución de un problema matemático difícil, a través de la manipulación de cadenas de ADN, en laboratorio. Para ello, consideró el problema del camino hamiltoniano, por ser un problema presuntamente intratable, y por haber sido objeto de exhaustivos estudios en la búsqueda de soluciones eficientes. En noviembre de 1994, consiguió resolver una instancia concreta del problema citado, en su versión dirigida y con un par de nodos distinguidos.

El experimento de L.M. Adleman se puede considerar como una prueba que prueba que algunas técnicas y herramientas de ingeniería genética proporcionan un nuevo marco para abordar la computabilidad de una forma muy atrayente.

4.5.2 Experimento de L.M. Adleman

Dado un grafo dirigido G = (V,A), y dos nodos distinguidos v_i, v_f, determinar si existe un camino hamiltoniano de G que va de v_i hasta v_f.

El camino se representara bajo una sucesión ordenada de vértices.

L.M Adleman resolvió el problema anterior para el grafo concreto con siete vértices y catorce aristas descrito en la siguiente figura, en donde los nodos distinguidos son 0 y 6, respectivamente.

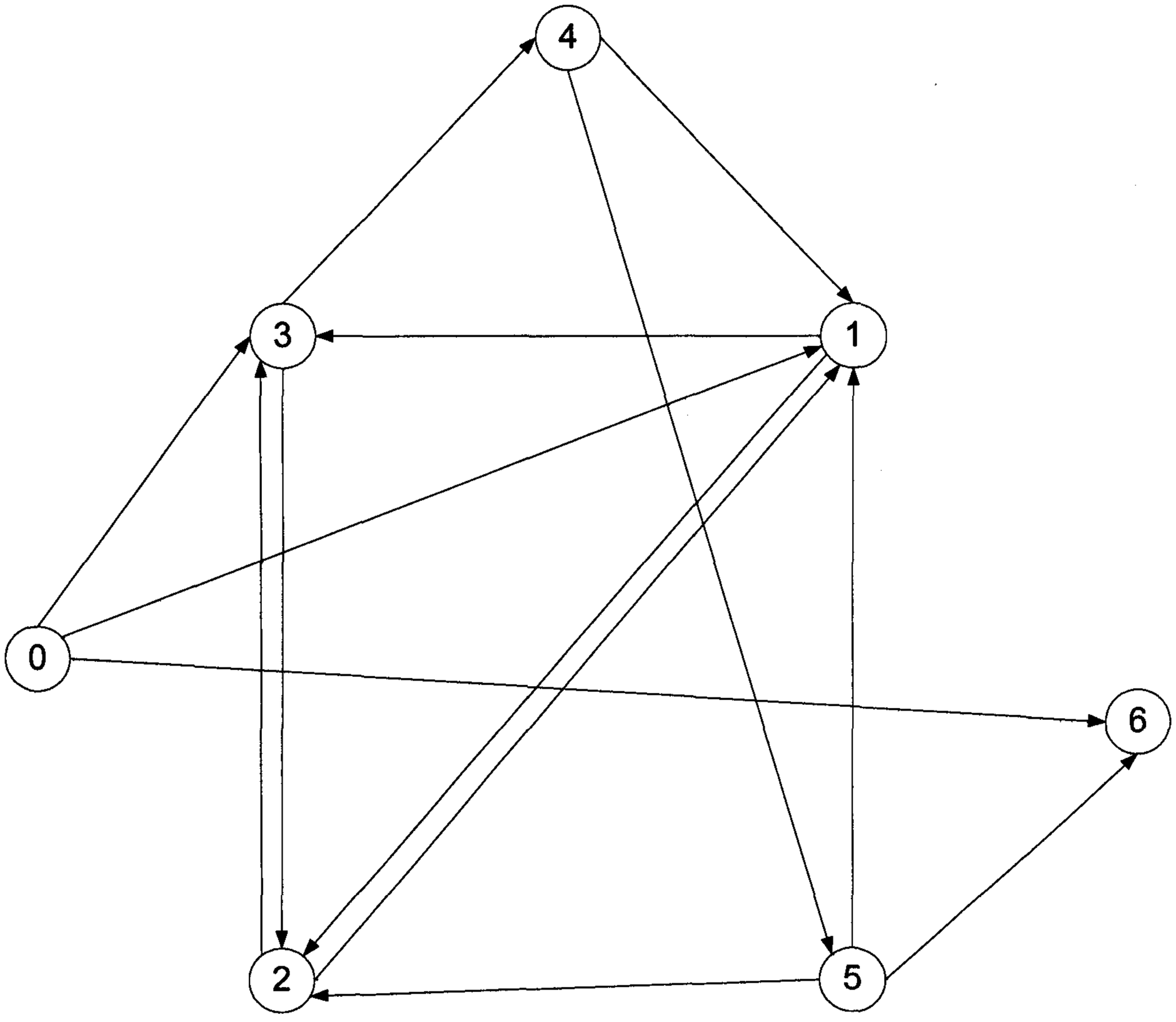


Figura 4.1.4. Grafo del experimento de Adleman.

Sólo existe en este grafo un camino hamiltoniano, que partiendo de 0, vaya a 1, 2 ,3 ,4 ,5 y 6, sucesivamente.

Como se puede comprobar esta solución puede ser obtenida por una persona en unos segundos y por un ordenador en menos de una millonésima de segundo. Sin

embargo, el experimento de Adleman necesitó siete días para su realización en el laboratorio.

Analizando un poco más detenidamente cómo se materializó dicho experimento, podemos deducir:

-Representa el primer ejemplo práctico de computación a nivel microscópico.

-Muestra el uso potencial de las moléculas de ADN como estructura de datos física, susceptibles de ser utilizadas como medio de almacenamiento, debido a la polaridad. Destaca en ellas la masiva densidad de información que pueden contener y el hecho de ser manipulables.

-Ilustra la posibilidad de usar las moléculas de ADN para resolver instancias de problemas de tipo combinatorio computacionalmente intratables.

-Muestra la capacidad que tienen las moléculas de ADN para simular computaciones de forma masivamente paralela, gracias a la acción simultánea de las enzimas sobre las cadenas de ADN en disolución, tal y como ocurren en el interior de los organismos vivos.

4.5.3 Implementación normal.

Se puede resolver el problema por fuerza bruta, a través del siguiente algoritmo de búsqueda exhaustiva:

Entrada: G, grafo dirigido; v_i y v_j nodos distinguidos.

1. Generar todos los caminos de G.
2. Rechazar los caminos que no empiezan por v_i y terminan por v_j.
3. Rechazar los caminos que no contienen todos los nodos.
4. Para cada u ϵ V, rechazar los caminos que no pasan por u.

Salida: SI, si queda algún camino; NO en otro caso.

Claro esta que el número de operaciones que conlleva este algoritmo son muy elevadas, debidamente explicitadas, reflejan un número exponencial de pasos simples, en función del tamaño del grafo.

4.5.4 Implementación molecular.

El experimento de Adleman, realiza básicamente una implementación directa del anterior algoritmo. A grandes rasgos, procede como sigue:

-Se codifican los vértices y los arcos del grafo mediante oligos, de tal manera que los caminos del grafo se pueden obtener a partir de dichos oligos realizando operaciones de naturalización y desnaturalización.

- A partir de un tubo de ensayo inicial que contiene moléculas de ADN que codifican cualquier posible camino de G, se van seleccionando las moléculas de acuerdo con los criterios indicados en cada uno de los pasos del algoritmo.

Al final de todos esos pasos, las moléculas de ADN permanezcan en el tubo de salida codificarán caminos hamiltonianos del grafo que van desde el nodo v_i hasta el nodo v_f.

El paso numero 1, plantea un problema en si mismo, generar todos los caminos de un grafo dirigido. En la solución de L.M. Adleman este paso se resuelve elaborando un tubo de ensayo inicial que contenga moléculas que codifican cualquier posible camino del grafo.

Para ello, a cada nodo del grafo, i ($0 \leq i \leq 6$), se le asocia un oligo, s_i, de longitud 20 mer (la longitud elegida depende realmente del número de elementos a codificar, debiendo ser suficiente para que no se produzcan enlaces no deseables y, a la vez, lo más pequeña posible para reducir tiempo de ejecución y facilitar su creación).

Para cada oligo se definirá un s'_i, que serán los primeros diez nucleótidos, y un s''_i que definirá los últimos diez nucleótidos (solo para este caso, la longitud de estos elementos será la mitad de la longitud de s_i). Así se han codificado cada vértice.

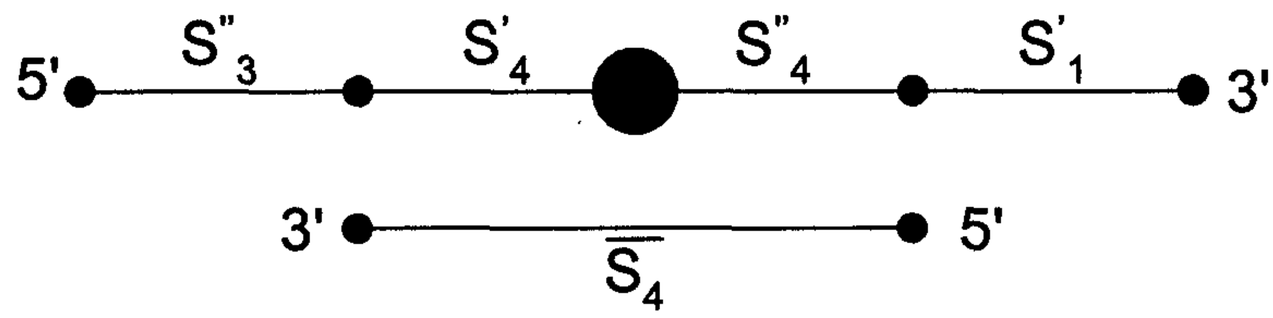
Ahora falta por codificar los arcos. Para cada arco (i, j) del grafo se le asocia el oligo (e_{ij}) formado por:

$$e_{ij} = \begin{cases} s_i'' s_j' \text{ si } i \neq 0 \wedge j \neq 6 \\ s_0 s_j' \text{ si } i = 0 \wedge j \neq 6 \\ s_i'' s_6 \text{ si } i \neq 0 \wedge j = 6 \\ s_0 s_6 \text{ si } i = 0 \wedge j = 6 \end{cases}$$

La idea para representar un camino del grafo (i_1, i_2,....i_k) consiste en considerar una disolución que contenga las cadenas s_{i1}, s_{i2}, ..., s_{ik} junto con los oligos asociados a los arcos $e_{i_1 i_2}, e_{i_2 i_3}$,...., $e_{i_{k-1} i_k}$ y someterla a un proceso de renaturalización.

Para elaborar el tubo inicial que contenga todos los posibles caminos del grafo G = (V, A), se parte de un tubo que contiene disolvente y se añade una cierta cantidad (Adleman usó 50 pmoles) de oligos s_i, para cada nodo i y la misma cantidad de oligos e_{ij} para cada arco (i,j) $\in$ A.

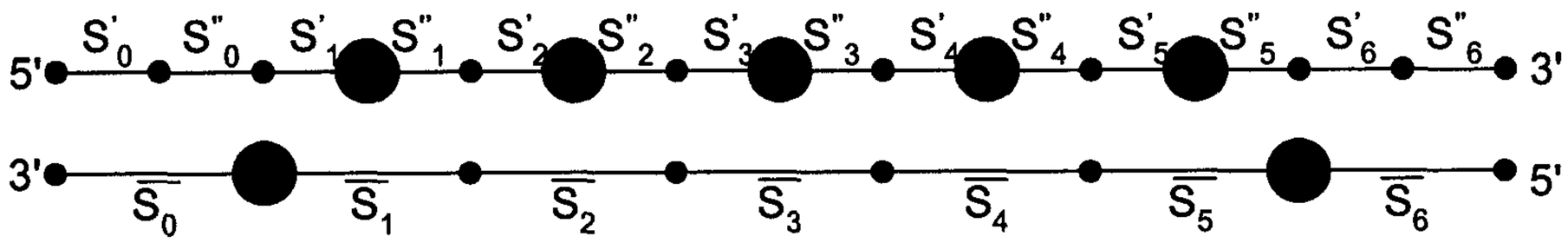
La solución obtenida se renaturaliza, usando la ligasa como enzima para rellenar los huecos que pudieran aparecer entre nucleótidos. De esta manera, se obtiene una serie de dobles hebras que codifican caminos del grafo. Más aun, las cantidades iniciales consideradas garantizan, con una probabilidad razonablemente alta, que cada camino del grafo aparecerá codificado en el tubo inicial así elaborado.



En la figura se ilustra este proceso con un ejemplo de un par de caminos del grafo.

Cuando se tienen todos los caminos posibles del grafo, comienza un proceso de filtrado, para ir rechazando los caminos que no cumplen con las condiciones necesarias para el camino hamiltoniano desde el vértice 0 hasta el 6.

Para implementar el paso 2 (rechazar todos los caminos que no empiezan por 0 y terminan en 6) se aplica la técnica de la reacción en cadena de la polimerasa, utilizando como bordes los oligos s_0 y s_6, respectivamente. De esta forma, sólo serán amplificados los caminos del grafo que comienzan por el vértice 0 y terminan por el vértice 6.



A continuación, se utiliza la técnica de la electroforesis en gel para seleccionar las moléculas de longitud 140. De esta manera se implementa el paso 3 (rechazar todos los caminos que no contienen exactamente 7 vértices).

Queda por implementar el ultimo paso, el cuatro,(para cada nodo del grafo, rechazar todos los caminos que no pasan por él) que se realiza de la siguiente forma: para cada i ($1 \leq i \leq 5$) se extraen las moléculas que contienen al oligo s_i (obsérvese que, por el paso dos, no es necesario considerar aquí los casos de i = 0 y i = 6).

Finalmente, para detectar si hay algún camino en el tubo de ensayo resultante, se amplifica el producto obtenido mediante la técnica PCR y se hacen circular las moléculas obtenidas a través de un gel.

4.5.5 Consideraciones.

-Las operaciones moleculares que tienen lugar en los tubos son aplicadas simultáneamente sobre todas las moléculas presentes en el mismo.

-En el tubo de ensayo inicial existe un número de cadenas que es exponencial en el tamaño del dato de entrada.

-La elección que hizo L.M. Adleman fue excesiva y aparecían los caminos pero con una cantidad bastante elevada. Esta cantidad debe depender de la densidad y del tamaño del grafo, pero en general, tiene que ser un número pequeño en este caso de 20 mer, con esta cantidad se garantiza que en el tubo inicial de ensayo estén todas las cadenas que codifican esos caminos y solo esos. También en el experimento los oligos de los nodos inicial y final eran de tamaño 30.

A pesar de que hay estudios con la cuestión de la elección de los oligos, ninguno es definitivo, es bastante complejo, ya que una buena elección debe evitar los enlaces no deseables entre trozos de cadenas que no son complementarias o entre distintas porciones de la misma cadena y, al mismo tiempo, no deben ser excesivamente grandes.

-En la ejecución del experimento aparecen errores debido a que las operaciones moleculares que se realizan no son perfectas: en algún caso se pueden seleccionar cadenas no deseadas, o bien rechazar alguna cadena que codifique una solución correcta. No obstante, estos errores pueden ser controlados y amortiguados, en cierta medida, a fin de conseguir los efectos deseados con una alta probabilidad.

- Varios autores como (D. Boneh, C. Dunworth y R.J. Lipton) conjeturan que el número máximo de moléculas de ADN que se pueden procesar en un experimento molecular es del orden de 10^{21}. De acuerdo con esta conjetura, el experimento de Adleman solo podría ser simulado para grafos de tamaño menor o igual a 70 (como curiosidad para simular un grafo de 200 nodos y densidad media, se necesitaría el volumen molecular equivalente al peso de la tierra.)

A pesar de todo lo dicho, se deben de resaltar las ventajas que proporciona la computación molecular en relación con el mejor supercomputador del momento:

-Velocidad de cálculo. La máxima velocidad de cálculo alcanzada en el experimento fue de $1,2 \times 10^{18}$ operaciones por segundo, mientras que el mejor supercomputador alcanza un máximo de 10^{12} operaciones por segundo.

-Consumo de energía. En el experimento, con 1 julio de trabajo se consiguió realizar hasta 2×10^{19} operaciones (por la segunda ley de termodinámica, el número máximo de operaciones por segundo es de 34×10^{19}), mientras que el mejor supercomputador es capaz de realizar como máximo 10^9 operaciones por julio.

-Densidad de almacenamiento de información. En el experimento se consiguió una densidad de almacenamiento de información de 1 bit por nanómetro cúbico, mientras que el mejor supercomputador de la actualidad tiene una densidad máxima de información del orden de 1 bit por 10^{12} nm^3 (un gramo de ADN ocupa en seco 1 centímetro cúbico, aproximadamente, y es capaz de almacenar la información equivalente a más de 1 billón de CD´s).

Desde el punto de vista computacional es necesario disponer de mecanismos que permitan atacar la resolución de problemas aplicando procedimientos de fuerza bruta susceptibles de ser implementados explotando el paralelismo masivo.

Si se encuentra un algoritmo adaptado a las peculiaridades de la estructura de datos que se manejan, se podría mejorar de manera ostensible los resultados obtenidos.

Computaciones NP-Completas pueden ser simuladas a través de una serie de protocolos que usan las moléculas de ADN como sustrato computacional. Esto abre nuevas perspectivas a la hora de mejorar cuantitativamente la resolución práctica de problemas NP-completos.

Para los biólogos pone en manifiesto la capacidad que tienen ciertos sistemas biológicos simples para simular procedimientos de cálculo muy complejos. Así se puede dar luz a mecanismos que subyacen en sistemas biológicos como el del sistema inmunológico o el de la evolución.

4.5.6 Conclusiones.

Desde el punto de vista computacional es necesario disponer de mecanismos que permitan atacar la resolución de problemas aplicando procedimientos de fuerza bruta susceptibles de ser implementados explotando el paralelismo masivo.

Si se encuentra un algoritmo especificicamente adaptado a las pecularidades de la estructura de datos que se manejan, se podría mejorar de manera ostensible los resultados obtenidos.

Computaciones NP-Completas pueden ser simuladas a través de una serie de protocolos que usan las moléculas de ADN como sustrato computacional. Esto abre

nuevas perspectivas a la hora de mejorar cuantitativamente la resolución práctica de problemas NP-completos.

4.6 Fundamentos de compilación

4.6.1 Analizador léxico

4.6.1.1 ¿Qué es un analizador léxico?

El analizador léxico se encarga de identificar y emitir tokens. Un token es un componente léxico o palabra. El analizador léxico tiene que ir buscando estos tokens desde el programa fuente. Los tokens son dados al analizador sintáctico como se indica en la figura 4.6.1.

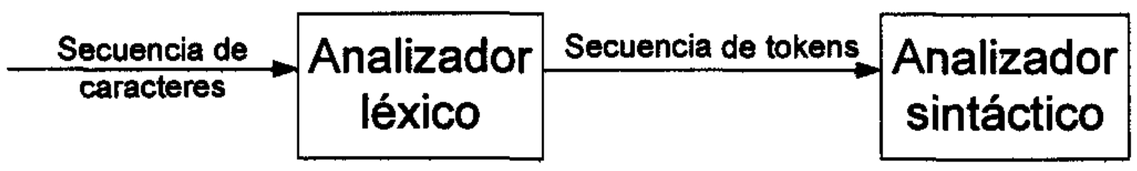


Figura 4.6.1. Entrada y salida del analizador léxico.

El token se genera mediante la definición de una gramática regular.

4.6.1.2 Funciones del analizador léxico

El analizador léxico es la primera fase de un compilador. Su principal función es leer los caracteres de entrada y elaborar como salida una secuencia de componentes léxicos que utilizará el analizador sintáctico para realizar su análisis.

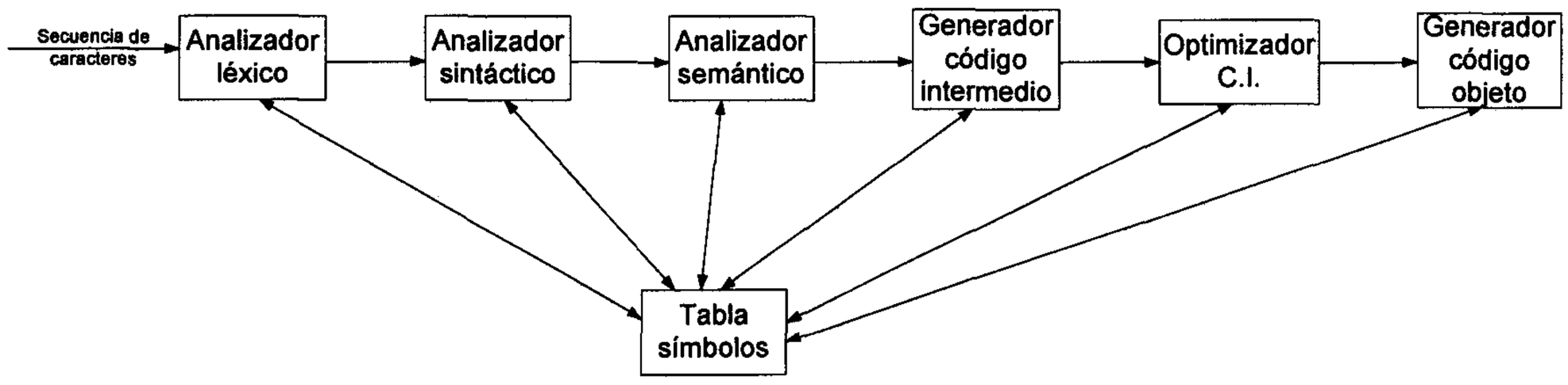


Figura 4.6.2. Interacción de llamadas y conectividad con Tabla de símbolos.

Otras funciones que realiza:

- Eliminar los comentarios del programa.
- Eliminar espacios en blanco y caracteres que no son necesarios para la sintaxis del lenguaje.
- Reconocer las variables e incluirlas en la tabla de símbolos.
- Llevar la cuenta de la posición por donde vamos trabajando. (línea, caracteres).

- Avisar de errores léxicos.
- Si el programa fuente incorpora metanociones o macros, el analizador léxico puede incorporar un preprocesador.
- Identificación e interpretación de números.

4.6.1.3 Necesidad del analizador léxico

Un tema importante es el porqué se separan los análisis léxico y sintáctico. Algunas razones de esta separación son:

- El diseño del análisis sintáctico es más fácil de esta forma, ya que éste no ha de preocuparse de leer el fichero de entrada, ni de saltar blancos, ni comentarios, ni de recibir caracteres inesperados, puesto que todo ello ha sido filtrado previamente por el analizador léxico. Por tanto, el diseño consiguientemente se hace más claro y comprensible.
- Se mejora la eficiencia del compilador. Un analizador léxico independiente permite construir un procesador especializado y potencialmente más eficiente para esa función. Gran parte del tiempo se consume en leer el programa fuente y dividirlo en componentes léxicos. Con técnicas especializadas de manejo de buffers para la lectura de caracteres de entrada y procesamiento de componentes léxicos se puede mejorar significativamente el rendimiento de un compilador.
- Aumenta la portabilidad del compilador, ya que todas las diferencias que se produzcan en el alfabeto de entrada, o en el dispositivo de almacenamiento, pueden ser reducidas al analizador léxico.
- Otra razón por la que se separan los dos análisis es para que el analizador léxico se centre en el reconocimiento de componentes básicos complejos.

4.6.1.4 Conceptos de tokens, patrones y lexemas

El analizador léxico puede tener la siguiente estructura:

(expresión regular 1) (acción 1)

(expresión regular 2) (acción 2)

...

(expresión regular n) (acción n)

Donde cada acción es un fragmento de programa que describe cual ha de ser la acción del analizador léxico cuando la secuencia de entrada coincida con la expresión regular.

Token o componente léxico. Es cada uno de las cadenas de caracteres del lenguaje con significado propio y que no se pueden subdividir.

Los tokens suelen ser las palabras claves y/o reservadas del lenguaje, operadores (aritméticos, relacionales y lógicos), constantes (enteros, reales, cadenas de caracteres,...) y signos de puntuación.

Patrón. Esta siempre asociado a un token. Es una regla que describe un conjunto de cadenas de la entrada para las cuales se produce como salida el mismo token.

Lexema. Secuencia de caracteres del programa fuente que concuerda con un patrón, es decir, es como una instancia de un patrón.

Una vez detectado que un grupo de caracteres coincide con un patrón, se ha detectado un lexema. A continuación se le asocia un número, que se le pasará al sintáctico, y, si es necesario, información adicional, como puede ser una entrada en la tabla símbolos.

4.6.1.5 Diseño de un analizador léxico.

Los pasos a seguir en le diseño de un análisis léxico son los siguientes:

1. Construir una gramática de tipo léxico, es decir, una gramática regular *tipo 3*.
2. Identificar los tokens del lenguaje.
3. A partir de la gramática construir un autómata finito determinista.
4. Incorporar al autómata una seria de acciones semánticas en forma de pseudocódigo.
5. Incluir los mensajes de error.

4.6.1.6 Acciones semánticas

Las acciones semánticas estás asociadas a las transiciones entre los estados del autómata finito y no a los propios estados. Las más comunes son:

LEE. Toma el siguiente carácter del fichero y lo deja disponible para realizar la siguiente transición.

BUSCA. Recibe una palabra como entrada y la busca en la tabla de símbolos y devuelve la posición de memoria en que la encuentra.

AÑADE. Recibe una palabra como entrada y la añade a la tabla de símbolos devolviendo la posición donde la metió.

GEN_TOKEN. Forma el token que le corresponde según el fichero de datos, lo envía al analizador sintáctico para que lo trate.

ERROR. Llama al gestor de errores porque se ha detectado un error.

4.6.1.7 Errores léxicos

Sólo hay errores léxicos al no detectar ningún patrón para la secuencia de entrada. Los errores léxicos típicos son:

-Caracteres que no pueden ser identificados.

-Falta de cierre de algún patrón, como las comillas para la definición de cadenas.

4.6.2 Tabla de Simbolos

Tiene dos funciones principales:

-Efectuar chequeos semánticos.

-Generación de código.

Permanece sólo el tiempo de compilación, no de ejecución, excepto en aquellos casos en que se compila con opciones de depuración.

La tabla almacena la información que en cada momento se necesita sobre las variables del programa, información tal como: nombre, tipo, dirección de localización, tamaño, etc... La gestión de la tabla de símbolos es muy importante, ya que consume gran parte del tiempo de compilación. De ahí que su eficiencia sea crítica. Aunque también sirve para guardar la información referente a los tipos creados por el usuario, tipos enumerador y, en general, a cualquier identificador, se centra principalmente en las variables de usuario. Respecto de cada una de ellas podemos guardar:

-Nombre.

Se puede hacer con o sin límite. Si lo hacemos con límite, emplearemos una longitud fija para cada variable, lo cual aumenta a velocidad de creación, pero limita la longitud en unos casos, y desperdicia espacio en la mayoría. Otro método es habilitar la memoria que necesitemos en cada caso para guardar el nombre.

-Tipo.

-Dirección de memoria.

Esta dirección es necesaria, porque las instrucciones que referencian a una variable deben saber donde encontrar el valor de esa variable en tiempo de ejecución, también cuando se trata de variables globales. En lenguajes que no permiten recursividad, las direcciones se van asignando secuencialmente a medida que se hacen las declaraciones. En lenguajes con estructuras de bloques, la dirección se da con respecto al comienzo del bloque de datos de ese bloque, en concreto.

-El número de dimensiones de una variable array, o el de parámetros de una función o un procedimiento junto con el tipo de cada uno de ellos es útil para el chequeo semántico. Aunque esta información puede extraerse de la estructura de tipos, para un control más eficiente, se puede indicar explícitamente.

-También se puede guardar información de los números de línea en los que se ha usado un identificador y en cual se declaró.

4.6.2.1 Consideraciones sobre la tabla de símbolos

Se debe de inicializar la tabla de símbolos con la siguiente información:

- Constantes: Pi, e,...
- Funciones de librería: EXP,...
- Palabras reservadas. Esto facilita el trabajo al analizador léxico, que tras reconocer un identificador lo busca en la tabla de símbolos, y si es palabra reservada devuelve el token asociado.

Conforme van apareciendo nuevas declaraciones de identificadores, el analizador léxico insertará nuevas entradas en la tabla de símbolos, evitando siempre la existencia de entradas repetidas.

La tabla de símbolos contiene información útil para poder compilar, por tanto existe en tiempo de compilación, y no de ejecución.

Sin embargo en un intérprete, dado que la compilación y ejecución se producen a la vez, la tabla de símbolos permanece todo el tiempo.

4.6.2.2 Funciones de la tabla de símbolos

Las funciones que se pueden realizar sobre la tabla de símbolos son las siguientes:

- Creación de nuevas entradas.
- Inserción de los atributos de una entrada.
- Búsqueda de una entrada.
- Acceso a la información de una entrada.

Estas operaciones han de estar implementadas eficientemente para que la tabla de símbolos no se convierta en el cuello de botella del compilador.

4.6.2.3 Tamaño de la tabla de símbolos

La tabla de símbolos puede tener un tamaño un fijo, aunque lo ideal es que sea dinámica para que vaya creciendo según se vayan añadiendo nuevas entradas.

4.6.2.4 Organización de a tabla de símbolos

- Tabla de símbolos lineal. Todas las entradas son consecutivas sin ningún tipo de organización, siendo el acceso a la misma secuencial. Es el método más fácil de implementar, pero también menos eficiente.

- Tabla de símbolos ordenada. Es una variante de la tabla de símbolos lineal en la que se ordenan los lexemas según se van insertando. La búsqueda en esta tabla es más eficiente aunque la inserción es más compleja. Si la tabla es dinámica, la inserción no se penaliza mucho.

- Tabla de símbolos hash. Consiste en determinar una función que transforma el lexema del identificador que se está buscando en la posición donde está almacenado. Deben evitarse las colisiones, es decir, que dos lexemas diferentes les corresponda una misma dirección. Para ello se dan tres tipos de tablas hash:

    - Tabla hash directa. Al producirse una colisión se busca el siguiente hueco en la tabla y ahí se inserta el nuevo identificador.

    - Tabla hash con overflow. Los identificadores que producen una colisión se llevan a otra tabla.

    - Tabla hash con encadenamiento. La tabla principal sólo contiene punteros a las tablas auxiliares en las que se almacena la información.

- Árbol binario ordenado. Árbol en el que el hijo izquierdo es menor que el nodo raíz y éste, a su vez, es menor que el hijo derecho.

La búsqueda se realiza recorriendo los nodos del árbol y comprobando si el lexema que se busca es mayor o menor que dicho nodo.

4.6.2.5 Ámbito de una variable

Un lenguaje suele poder contener variables globales y locales, debido a esto se tiene que tratar la tabla de símbolos de forma especial, para que los ámbitos de las variables no sean confundidos. Por ejemplo: llamar a una variable local, en ámbito global.

Para solucionarlo se siguen varias reglas:

- Sólo son accesibles los nombres definidos en ámbito actual y en los ámbitos abiertos que le rodean.

- Sin un nombre se declara en más de un ámbito abierto, sólo es accesible el que está definido en el ámbito más profundo.

Hay dos formas para implementarlo, son:

- Una tabla por ámbito.
- Una tabla única para todos los ámbitos.

La tabla única suele utilizarse en compiladores de un solo paso, en los que se puede descartar la información referente a un ámbito en cuanto se cierra. Un compilador de múltiples pasos suele usar una tabla individual por ámbito.

4.6.2.6 Tabla de símbolos por ámbito

Hay que asegurarse de que sólo son accesibles los nombres que cumplen las reglas. Para ello se crea una lista de ámbitos. Los ámbitos abiertos se encuentran situados en la lista en orden de apertura, con el actual en primer lugar. El problema surge con el manejo de los ámbitos cerrados. En este caso hay que tener en cuenta el tipo de compilador:

- Si el compilador es de un solo paso, los ámbitos cerrados se pueden descartar, y la lista se convierte en una pila. Para buscar un nombre se empieza por el ámbito situado en lo alto de la pila y se va descendiendo hasta que se encuentra o hasta que se acaba la pila.
- Si el compilador tiene más de un paso, hará falta guardar la información relativa a los ámbitos cerrados para utilizarla en pasos posteriores.

La búsqueda se realiza sólo en el bloque actual si se trata de una declaración, o en él y sus antepasados en cualquier otro caso.

Los problemas que presenta esta implantación son:

- Búsqueda en múltiples tablas de símbolos, con la consiguiente pérdida de tiempo.
- Fragmentación del espacio de almacenamiento de las tablas.

4.6.2.7 Tabla de símbolos única

En este caso, todos los identificadores de todos los ámbitos están en la misma tabla. Cada identificador lleva información sobre el ámbito al que pertenece. Un identificador puede aparecer varias veces, siempre que cada aparición lleve un número de ámbito diferente.

- **Procedimiento de creación.** El final de la tabla de símbolos se usa como una lista al revés. Cuando aparece el comienzo de un nuevo bloque, se pone el puntero al final de la lista. Cuando se llega al final de un bloque,

se vacía la lista correspondiente a ese bloque, copiando al principio de la tabla.

- **Inserción.** Se añade a la lista del final de la tabla de símbolos, se corre el puntero y se aumenta en uno el número de elementos de ese bloque.

- **Búsqueda**. Se busca en el bloque presente sólo (si es una declaración) o en él y sus antepasados en otro caso.

Las ventajas de esta implantación respecto a la de múltiples tablas son:

- La búsqueda es más rápida, ya que sólo hay que buscar en una tabla.

- El espacio ocupado es menor, pero la diferencia queda compensada con el hecho de que en esta implantación hay un campo más (ámbito).

4.6.3 Análisis sintáctico

Todo lenguaje de programación tiene reglas que describen la estructura sintáctica de programas bien formados. Se puede describir la sintaxis de las construcciones de los lenguajes de programación por medio de gramáticas de contexto libre o notación BackusNaur Form (BNF).

Las gramáticas ofrecen ventajas significativas a los diseñadores de lenguajes y a los desarrolladores de compiladores.

- Las gramáticas son especificaciones sintácticas y precisas de lenguajes de programación.

- A partir de una gramática se puede generar automáticamente un analizador sintáctico.

- El proceso de construcción puede llevar a descubrir ambigüedades.

- Una gramática proporciona una estructura a un lenguaje de programación, siendo más fácil generar código y detectar errores.

- Es más fácil ampliar/modificar el lenguaje si está descrito con una gramática.

4.6.3.1 ¿Qué es el análisis sintáctico?

Es la fase del analizador que se encarga de chequear el testo de entrada en base a una gramática dada. Y en caso de que el programa de entrada sea válido, suministra el árbol sintáctico que lo reconoce.

En teoría, se supone que la salida del analizador sintáctico es alguna representación del árbol sintáctico que reconoce la secuencia de tokens suministrada por el analizador léxico.

En la práctica, el analizador sintáctico también hace:

- Acceder a la tabla de símbolos (para hacer parte del trabajo del analizador semántico).
- Chequeo de tipos (del analizador semántico).
- Generar código intermedio.
- Generar errores cuando se producen.

En definitiva, realiza casi todas las operaciones de la compilación. Este método de trabajo da lugar a los métodos de compilación dirigidos por la sintaxis.

4.6.3.2 Manejo de errores sintácticos

Si un compilador tuviera que procesar sólo programas correctos, su diseño e implantación se simplificarían mucho. Pero los programadores a menudo escriben programas incorrectos, y un buen compilador debe ayudar al programador a identificar y localizar errores. Es más, considerar desde el principio el manejo de errores puede simplificar la estructura de un compilador y mejorar su respuesta a los errores.

Los errores en la programación pueden ser de los siguiente tipos:

- Léxicos, producidos al escribir mal un identificador, una palabra clave o un operador.
- Sintácticos, por una expresión aritmética o paréntesis no equilibrados.
- Semánticos, como un operador aplicado a un operando incompatible.
- Lógicos, puede ser una llamada infinitamente recursiva.

El manejo de errores de sintaxis es el más complicado desde el punto de vista de la creación de compiladores. Nos interesa que cuando el compilador encuentre un error, se recupere y siga buscando errores. Por lo tanto el manejador de errores de un analizador sintáctico debe tener como objetivos:

- Indicar los errores de forma clara y precisa. Aclarar el tipo de error y su localización.
- Recuperarse del error, para poder seguir examinando la entrada.

- No ralentizar significativamente la compilación.

Un buen compilador debe hacerse siempre teniendo también en mente los errores que se pueden producir; con ello se consigue:

- Simplificar la estructura del compilador.

- Mejorar la respuesta ante los errores.

Tenemos varias estrategias para corregir errores, una vez detectados:

- ✓ Ignorar el problema (Panic mode). Consiste en ignorar el resto de la entrada hasta llegar a una condición de seguridad. Una condición tal se produce cuando nos encontramos un token especial (por ejemplo un ; o un END). A partir de este punto se sigue analizando normalmente.

- ✓ Recuperación a nivel de frase. Intenta recuperar el error una vez descubierto. Hay que tener cuidado con este método, pues puede dar lugar a recuperaciones infinitas.

- ✓ Reglas de producción adicionales para el control de errores. La gramática se puede aumentar con las reglas que reconocen los errores más comunes.

- ✓ Corrección global. Dada una secuencia completa de tokens a ser reconocida, si hay algún error por el que no se puede reconocer, consiste en encontrar la secuencia completa más parecida que sí se pueda reconocer. Es decir, el analizador sintáctico le pide toda la secuencia de tokens al léxico, y lo que hace es devolver lo más parecido a la cadena de entrad pero sin errores, así como el árbol que lo reconoce.

4.6.3.3 Gramática que acepta un analizador sintáctico.

La gramática que acepta el analizador sintáctico es una gramática de contexto libre:

-Gramática. G(N,T,P,S)

N = No terminales.

T = Terminales.

P = Reglas de producción.

S = Axioma inicial.

- Derivaciones. La idea central es que se considera una producción como una regla de reescritura, donde el no terminal de la izquierda es sustituido por la cadena del lado derecho de la producción.
    - Derivación por la izquierda. Derivación donde solo el no terminal de más a la izquierda de cualquier forma de frase se sustituye en cada paso.
    - Derivación por la derecha o Derivación canónica. Derivación donde el no terminal más a la derecha se sustituye en cada paso.

4.6.3.4 Árbol sintáctico

Es una representación gráfica donde aparecen las producciones de la gramática aplicadas y en el orden que son aplicadas para obtener una secuencia de símbolos de un lenguaje.

Como nodos internos del árbol, se sitúan los elementos no terminales de las reglas de producción que vayamos aplicando, y tantos hijos como símbolos existan en la parte derecha de dichas reglas.

4.6.3.5 Gramática ambigüa

Una gramática que admita más de un árbol sintáctico para una misma secuencia de símbolos de entrada.

4.6.3.6 Gramática en forma normal de Chomsky

Toda gramática de contexto libre cuyas producciones son de la forma:

$A \rightarrow BC$

$A \rightarrow a$

donde A,B son símbolos no terminales y a y c son terminales.

4.6.3.7 Gramática en forma normal de Greibach

Toda gramática de contexto libre cuyas producciones son de la forma:

$A \rightarrow a\,\alpha$

donde A es símbolo no terminal, a es símbolo terminal y α es una cadena de símbolos no terminales y puede ser también la cadena vacía.

4.6.3.8 Tipos de análisis

De la forma de construir el árbol sintáctico se desprenden dos tipos o clases de analizador sintáctico. Pueden ser descendentes o ascendentes.

-Descendentes. Parten del axioma inicial, y van efectuando derivaciones a izquierda hasta obtener la secuencia de derivaciones que reconoce a la sentencia.

Pueden ser:

-Con retroceso.

-Con recursión.

-LL(1).

-Ascendentes. Parten de la sentencia de entrada, y van aplicando reglas de producción hacia atrás (desde el consecuente hasta el antecedente), hasta llegar al axioma inicial.

Pueden ser:

-Con retroceso.

-LR(1).

4.6.4 Análisis semántico

4.6.4.1 Gramáticas atribuidas

El análisis semántico es debido a que se tiene que comprobar que las reglas de la gramática tienen algún sentido. El chequeo semántico se encarga de que los tipos sean correctos, por ejemplo no se puede dividir un entero entre una cadena de caracteres.

En esta parte del compilador se desarrolla la traducción de lenguajes guiada por gramáticas de contexto libre. Se asocia información a una construcción del lenguaje de programación proporcionando atributos a los símbolos de la gramática que representan la construcción. Los valores de los atributos se calculan mediante "reglas semánticas" asociadas a las reglas de producción gramatical.

Hay dos notaciones para asociar reglas semánticas con reglas de producción, las Definiciones Dirigidas por Sintaxis (DDS) y Esquemas de Traducción (ET).

Las DDS son especificaciones de alto nivel para traducciones. No es necesario que el usuario especifique explícitamente el orden en el que tiene lugar la traducción.

Los ET indican el orden en que se deben evaluar las reglas semánticas.

Conceptualmente, tanto con las DDS como con los ET, se analiza sintácticamente la cadena de componentes léxicos de entrada, se construye el árbol de análisis sintáctico y después se recorre el árbol para evaluar las reglas semánticas en sus nodos. La evaluación de las reglas semánticas puede generar código, guardar información en una tabla de símbolos, emitir mensajes de error o realizar otras actividades. La traducción de la cadena de componentes léxicos es el resultado obtenido al evaluar las reglas semánticas.

NO todas las implementaciones tienen que seguir al pie de la letra este esquema. Hay casos especiales de DDS que se pueden implementar en una sola pasada evaluando las reglas semánticas durante el análisis sintáctico, sin construir explícitamente un árbol de análisis sintáctico o un grafo que muestre las dependencias entre los atributos.

4.6.4.2 Definiciones dirigidas por sintaxis

Una DDS es una gramática de contexto libre en la que cada símbolo gramatical (terminales y no terminales) tiene un conjunto de atributos asociados, dividido en dos subconjuntos llamados atributos sintetizados y atributos heredados de dicho símbolo gramatical. Si se considera un nodo de un símbolo gramatical de un árbol de análisis sintáctico como un registro con campos para guardar información, entonces un atributo corresponde al nombre de un campo.

Un atributo puede representar cualquier cosa: una cadena, un número, un tipo, una posición de memoria, etc... El valor de un atributo en un nodo de un árbol de

análisis sintáctico se define mediante una regla semántica asociada a la regla de producción utilizada en dicho nodo. El valor de un atributo sintetizado de un nodo se calcula a partir de los valores de los atributos hijos de dicho nodo en el árbol de análisis sintáctico; el valor de un atributo heredado se calcula a partir de los valores de los atributos en los hermanos y el padre de dicho nodo.

Las reglas semánticas establecen las dependencias entre los atributos que serán representadas mediante un grafo. Del grafo de dependencias se obtiene un orden de evaluación de las reglas semánticas. La evaluación de las reglas semánticas define los valores de los atributos en los nodos del árbol de análisis sintáctico para la cadena de entrada. Una regla semántica también puede tener efectos colaterales, por ejemplo, imprimir un valor o actualizar una variable global. Por supuesto, una aplicación no necesita construir explícitamente un árbol de análisis sintáctico o un grafo de dependencias; sólo tiene que producir el mismo resultado para cada cadena de entrada.

Un árbol de análisis sintáctico que muestre los valores de los atributos en cada nodo se denomina un árbol de análisis sintáctico con anotaciones. El proceso de calcular los valores de los atributos en los nodos se denomina anotar o decorar el árbol de análisis sintáctico.

4.6.4.3 Forma de una definición dirigida por sintaxis

En una DDS, cada regla de producción A → α tiene asociado un conjunto de reglas semánticas de la forma b:= f(c_1,c_2,...,c_k), donde f es una función, y, o bien

- b es un atributo sintetizado de A y c_1,c_2,...,c_k son atributos de los símbolos de α, o bien

- b es un atributo heredado de uno de los símbolos de α, y c_1,c_2,...,c_k son atributos de los restantes símbolos de α o bien de A.

En cualquier caso, se dice que el atributo b depende de los atributos c_1, c_2, ..., c_k.

Una gramática con atributos es una DDS en la que las funciones en las reglas semánticas no pueden tener efectos colaterales.

Las funciones de las reglas semánticas a menudo se escribirán como expresiones. Ocasionalmente el único propósito de una regla semántica en una DDS es crear un efecto colateral. Dichas reglas semánticas se escriben como llamadas a procedimientos o fragmentos de programa. Se pueden considerar como reglas que definen los valores de atributos sintetizados ficticios del no terminal del lado izquierdo de la regla de producción asociada; no se muestran el atributo ficticio y el signo := de la regla semántica.

Las reglas semánticas se ejecutan en el momento en que se reduce por su regla asociada.

En una DDS, se asume que los terminales sólo tienen atributos sintetizados, ya que la definición no proporciona ninguna regla semántica para los terminales. El AL es el que proporciona generalmente los valores de los atributos de los terminales. Además se asume que el símbolo inicial no tiene ningún atributo heredado, a menos que se indique lo contrario.

4.6.4.4 Atributos sintetizados

Los atributos sintetizados son muy utilizados en la práctica. Una DDS que usa atributos sintetizados exclusivamente se denomina definición con atributos sintetizados. Siempre se puede anotar un árbol de análisis sintáctico para una definición con atributos sintetizados mediante la evaluación de las reglas semánticas para los atributos en cada nodo de forma ascendente.

4.6.4.5 Atributos heredados

Un atributo heredado es uno cuyo valor en un nodo de un árbol de análisis sintáctico está definido a partir de los atributos en el padre y/o de los hermanos de dicho nodo. Los atributos heredados sirven para expresar la dependencia de una construcción de un lenguaje de programación en el contexto en el que aparece. Por ejemplo, se puede utilizar un atributo heredado para comprobar si un identificador aparece en el lado izquierdo o en el derecho de una asignación para decidir si se necesita la dirección o el valor del identificador. Aunque siempre es posible reescribir una DDS para que sólo se utilicen atributos sintetizados, a veces es más natural utilizar DDS con atributos heredados.

4.6.4.6 Grafo de dependencias

Si un atributo b en un nodo de un árbol de análisis sintáctico depende de un atributo c, entonces se debe evaluar la regla semántica para b en ese nodo después de la regla semántica que define a c. Las interdependencias entre los atributos heredados y sintetizados en los nodos de un árbol de análisis sintáctico se pueden representar mediante un grafo dirigido llamado grafo de dependencias.

Antes de construir un grafo de dependencias para un árbol de análisis sintáctico, se escribe cada regla semántica en la forma b := f (c_1, c_2, ..., c_k), introduciendo un falso atributo sintetizado b para cada regla semántica que conste de una llamada de procedimiento. El grafo tiene un nodo por cada atributo y una arista al nodo de b desde el nodo de c si el atributo b depende del atributo c. Más detalladamente, el grafo de dependencias para un determinado árbol de análisis sintáctico se construye de la siguiente manera:

```
for cada nodo n en el árbol de análisis sintáctico do
        for cada atributo a del símbolo gramatical en el nodo do
                construir un nodo en el grafo de dependencias para a;
for cada nodo n en el árbol de análisis sintáctico do
        for cada regla semántica b:=f(c_1,c_2,...,c_k) asociada con la producción
                utilizada en nodo
                for i:= 1 to k do
                        construir una arista desde el nodo para c_i hasta el nodo
                        para b;
```

Por ejemplo, supóngase que A.a:= f (X.x, Y.y) es una regla semántica para la producción A --> XY. Esta regla define un atributo sintetizado A.a que depende de los atributos X.x y Y.y. Si se utiliza esta producción en el árbol de análisis sintáctico, entonces habrá tres nodos, A.a, X.x e Y.y, en el grafo de dependencias con una arista desde X.x hasta A.a puesto que A.a depende de X.x y una arista desde Y.y hasta A.a puesto que A.a también depende de Y.y.

Si la regla de producción A → XY tiene asociada la regla semántica S.i:= g (A.a,Y.y). Entonces habrá una arista desde A.a hasta X.i y también una arista desde Y.y hasta X.i, puesto que X.i depende tanto de A.a como de Y.y.

4.6.4.7 Orden de evaluación

Un ordenamiento topológico de un grafo dirigido acíclico es todo ordenamiento m_1, m_2, ..., m_k de los nodos del grafo tal que las aristas vayan desde los nodos que aparecen primero en el ordenamiento a los que aparecen más tarde; es decir, si m_i → m_j es una arista desde m_i a m_j, entonces m_i aparece antes que m_j en el ordenamiento.

Todo ordenamiento topológico de un grafo de dependencias da un orden válido en el que se pueden evaluar las reglas semánticas asociadas con los nodos de un árbol de análisis sintáctico. Es decir, en el ordenamiento topológico, los atributos dependientes c_1, c_2,..., c_k en una regla semántica b := f(c_1, c_2, ..., c_k) están disponibles en un nodo antes de que se evalúe f.

La traducción especificada por una DDS se puede precisar como sigue. Se utiliza la gramática subyacente para construir un árbol de análisis sintáctico para la entrada. El grafo de dependencias se construye como se indico anteriormente. A partir de un ordenamiento topológico del grafo de dependencias, se obtiene un orden de evaluación para las reglas semánticas. La evaluación de las reglas semánticas en este orden produce la traducción de la cadena de entrada.

4.6.4.8 Esquemas de traducción

Un ET es una gramática de contexto libre en la que se encuentran intercalados, en el lado derecho de la regla de producción, fragmentos de programas a los que hemos

llamado acciones semánticas. Un esquema de traducción es como una definición dirigida por sintaxis, con la excepción de que el orden de evaluación de las reglas semánticas se muestra explícitamente. La posición en la que se ejecuta alguna acción se da entre llaves y se escribe en el lado derecho de una producción.

A → cd {printf (c+d)} B

Cuando se diseña un ET, se deben respetar algunas limitaciones para asegurarse de que el valor de un atributo esté disponible cuando una acción se refiera a él. Estas limitaciones, motivadas por las definiciones dirigidas por sintaxis, garantizan que las acciones no hagan referencia a un atributo que aún no haya sido calculado.

El ejemplo más sencillo ocurre cuando sólo se necesitan atributos sintetizados. En este caso, se puede construir el ET creando una acción que conste de una asignación para cada regla semántica y colocando esta acción al final del lado derecho de la regla de producción asociada. Por ejemplo, la producción T → T * F y la regla semántica T.val:= T.val * F.val dan como resultado la siguiente producción y acción semántica:

T → T1 * F T.val:= T1.val * F.val

Si se tienen atributos tanto heredados como sintetizados, se debe ser más prudente:

1. Un atributo heredado para un símbolo en el lado derecho de una regla de producción se debe calcular en una acción antes que dicho símbolo.

2. Una acción no debe referirse a un atributo sintetizado de un símbolo que esté a la derecha de la acción.

3. Un atributo sintetizado para el no terminal de la izquierda sólo se puede calcular después de que se hayan calculado todos los atributos a los que hace referencia.

La acción que calcula dichos atributos se puede colocar generalmente al final del lado derecho de la producción.

4.6.5 Generación de código intermedio

En el modelo de análisis y síntesis de un compilador, la etapa inicial traduce un programa fuente a una representación intermedia a partir de la cual la etapa final genera el código objeto. Los detalles del lenguaje objeto se confinan en la etapa fina., si esto es posible. Aunque un programa fuente se puede traducir directamente al lenguaje objeto, algunas ventajas de utilizar una forma intermedia independiente de la máquina son:

- Se facilita la redestinación; se puede crear un compilador para una máquina distinta uniendo una etapa final para la nueva máquina a una etapa inicial ya existente.

- Se puede aplicar a la representación intermedia un optimizador de código independiente de la máquina.

Hay lenguajes que son pseudointerpretados que utilizan un código intermedio llamado código-P que utiliza lo que se denomina bytecodes (sentencias de un μP hipotético). Por ejemplo Java utiliza los ficheros .class, éstos tienen unos bytecodes que se someten a una JavaVirtualMachine, para que interprete esas sentencias.

En este capítulo se muestra cómo se pueden utilizar loso métodos de DDS para traducir a un código intermedio, construcciones de lenguajes de programación como declaraciones, asignaciones y proposiciones de flujo de control. La generación de código intermedio se puede intercalar en el análisis sintáctico.

4.6.5.1 Tipos y Representaciones del código intermedio

Existen muchas formas de código intermedio, de hecho, el diseñador del compilador puede definir la máquina abstracta que considere mas adecuada al lenguaje fuente o a la clase de máquinas a las que va destinado. Las representaciones que más se emplean son:

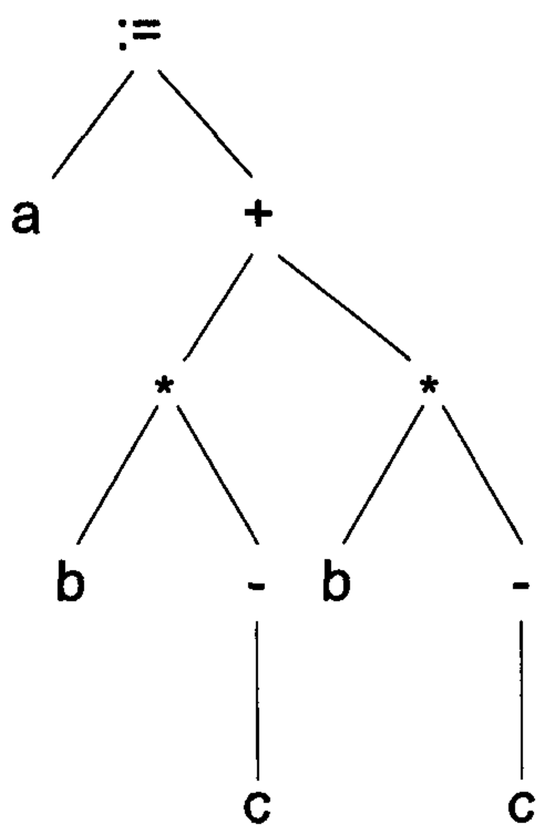
- Árboles semánticos y grafos acíclicos dirigidos.

- Código de tres direcciones: cuartetos, tercetos o tercetos indirectos.

4.6.5.2 Árboles semánticos y grafos acíclicos dirigidos

Una forma de representar el código generado por un compilador es mediante una estructura de tipos arborescente, a la que se denomina árbol semántico a fin de diferenciarlo del árbol sintáctico que representa la estructura gramatical.

Así, por ejemplo, para la representación de la secuencia: a:= b* -c + b * -c, se tiene la figura 4.6.5.1:



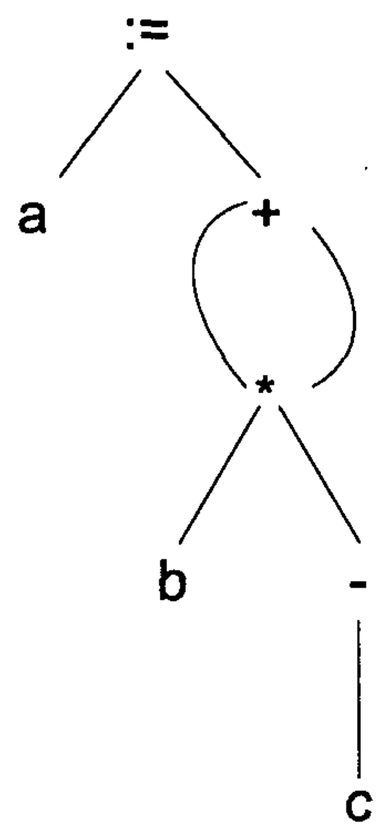


Figura 4.6.5.1. Secuencia a:= b * -c + b * -c. **Figura 4.6.5.2. Secuencia reducida.**

En el árbol semántico los nodos son ocupados por operadores y sus operandos se obtienen evaluando las operaciones de sus nodos descendentes.

Pueden usarse Grafos Acíclicos Dirigidos (DAG) para obtener una representación condensada de árboles semánticos (Figura 4.6.5.2). Cuando se encuentran subestructuras idénticas dentro de un mismo árbol basta usar múltiples referencias a la misma subestructura, sin necesidad de duplicar la subestructura repetida. Esta representación consigue por si misma una mejora considerable en el tamaño y la eficiencia del código generado.

4.6.5.3 Código de tres direcciones

La representación mediante código de tres direcciones se basa en un conjunto de instrucciones capaces de manejar un máximo de tres direcciones de memoria. En general dos de estas direcciones corresponden a los argumentos y la tercera al resultado. Una instrucción de tres direcciones, por ejemplo, puede consistir en sumar el contenido de dos direcciones y situar el resultado en una tercera dirección, o saltar a una determinada dirección si el valor contenido en una dirección es mayor que el contenido en una dirección es mayor que el contenido en otra, etc...

Usando un adecuado conjunto de instrucciones de este tipo, cualquier sentencia o conjunto de sentencias del código fuente, puede traducirse por una secuencia de instrucciones de tres direcciones.

Esta representación es equivalente a la representación mediante árbol semántico o mediante DAG que se ha visto anteriormente, siempre que se limite la ramificación del árbol a un máximo de dos descendientes. Para obtener la representación en código de tres direcciones basta con recorrer el árbol ascendentemente de izquierda a derecha y generar una variable temporal para cada nodo intermedio del árbol, resumiendo mediante ella la estructura descendiente.

El conjunto de instrucciones de tres direcciones del código intermedio debe definirse adecuadamente según el lenguaje a compilar, y teniendo en cuenta que cada instrucción del mismo debe convertirse fácilmente en una o varias instrucciones del código máquina.

- Instrucciones de asignación
  - a:= b (directa)
  - a:= op b (de un operando)
  - a:= b op c (de donde operandos)
- Saltos de ejecución.
  - goto L (incondicional)
  - if a opcond b then goto L (condicional)
  - label L (etiqueta)
- Inserción y toma de parámetros y llamada y retorno a procedimientos y funciones.
  - param X (inserción de parámetros en la pila)
  - pop X (toma de parámetros en la pila)
  - call p, n(llamada a funciones y procedimientos)
  - return X (retorno de funciones)

Por lo general, se usan conjuntamente varias instrucciones de tres direcciones para realizar la llamada a un procedimiento o función. Primero se definen las variables que van a usarse como parámetros y luego se transfiere el control mediante una instrucción de llamada, indicando la dirección del procedimiento y el número de parámetros.

- Operaciones de direccionamiento indirecto.
  - x:= y[i] (en el argumento)
  - x[i] := y (en el resultado)
- Operaciones con punteros.
  - x:= &y (asignación de dirección)
  - x:= *y (asignación de valor)
  - *x:= y (asignación indirecta)

La elección de operadores permisibles es un aspecto importante en el diseño de código intermedio. El conjunto de operadores debe ser lo bastante rico como para implementar las operaciones del lenguaje fuente. Un conjunto de operadores pequeño es más fácil de implementar en una nueva máquina objeto. Sin embargo, un conjunto de instrucciones limitado puede obligar a la etapa inicial a generar largas secuencias de proposiciones para algunas operaciones del lenguaje fuente. En tal caso, el optimizador y el generador de código tendrán que trabajar más si se desea producir un buen código.

Cuando se genera código de tres direcciones, se construyen variables temporales para los nodos interiores del árbol sintáctico. Como en la figura 4.6.5.3, para el ejemplo

a = 3 + b + 5 + c

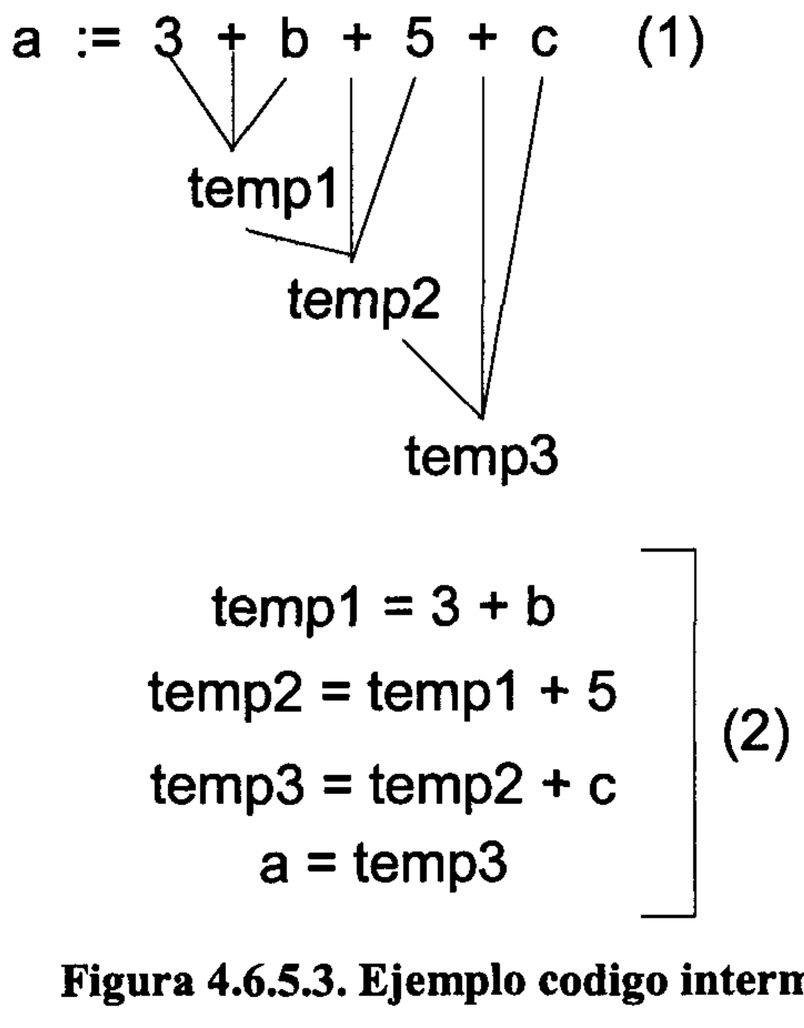


Figura 4.6.5.3. Ejemplo codigo intermedio.

La pregunta es ¿(1) y (2) son equivalentes? Sí, y además (2) es muy parecido a un código máquina.

Lo importante es saber hacer la reducciones y tener una gramática adecuada. El no terminal tiene que tener unos atributos asociados que los representen. Figura 4.6.5.4.

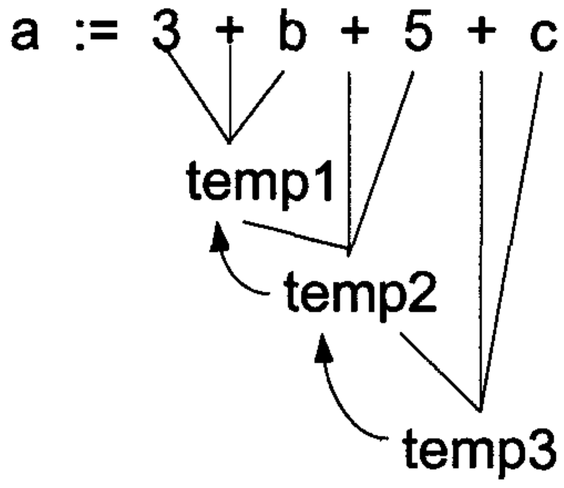


Figura 4.6.5.4. Dependencia entre variables temporales.

- temp1 representa a 3 + b
- temp2 representa a temp1 + 5
- temp3 representa a temp2 + c

Hay que saber qué representa cada atributo y como puedo generar código.

Según el problema que se plantee hay que ver si hace falta o no una tabla de símbolos. En el ejemplo anterior no hace falta una tabla de símbolos, porque no hay chequeo de tipos. Pero, por ejemplo, en un problema en el que exista una zona de declaración y una zona de definición, entonces sí haría falta tener una tabla de símbolos, o por ejemplo, si decimos que el contenido de la dirección de memoria 12h = a, entonces también se necesitaría una tabla de símbolos.

La representación que se ha utilizado hasta el momento se llama de cuartetos, ya que utiliza cuatro campos para almacenar cada instrucción. Existe otra técnica que permite representar el código de tres direcciones mediante solo tres campos. Para ello se mantiene la secuencia de instrucciones de código intermedio en un vector. Ya que toda instrucción lleva aparejada una dirección como resultado se hará mediante el número correspondiente a la posición del vector en la que se calculó. Por tanto, esta representación sólo almacena los campos correspondientes a la instrucción y a los dos argumentos y se llama de tercetos.

El uso de tercetos tiene una evidente ventaja sobre los cuartetos, ya que no es necesario generar un sinfín de variable temporales. Además es más compacta y por tanto ocupa menos espacio en memoria. Por contra, la representación mediante tercetos es más complicada y puede dificultar la optimización de código, ya que cualquier variación en la secuencia de instrucciones obliga a un reposicionamiento de las referencias. Este problema puede solventarse usando la representación de tercetos

indirectos, que consiste en utilizar un vector auxiliar para almacenar la secuencia de instrucciones.

4.6.5.4 Generación de código de tres direcciones en sentencias de control

Utilizando código de tres direcciones, se puede generar el código correspondiente no sólo a expresiones, sino también el correspondiente a sentencias de control.

En el caso de las expresiones, se usaban en las variables temporales para generar código. Ahora el problema son los cambios de flujos.

- IF-THEN-ELSE
- CASE
- WHILE
- REPEAT

4.6.5.4.1 Sentencia IF-THEN-ELSE

En este caso, la instrucción de entrada sería:

IF cond THEN
sent1
ELSE
sent2
END_IF

¿Cómo se genera el código de tres direcciones para una instrucción? La clave de todo está en la condición.

- Condición simple. Condición de la forma exp1 op exp2

Tendrá asociada en el esquema de traducción un par de atributos que serán dos etiquetas: una a la que se saltará en caso de que la condición se cierta (etq_verdad), y otra cuando la condición sea falsa (etq_falso). Así se tendrá:

cond: exp1 op exp2 *{cond.etq_verdad := nueva_etiqueta();*
cond.etq_falso := nueva_etiqueta();
escribe (if exp1.s op.s exp2.s goto cond.etq_verdad);
escribe (goto cond.etq_falso)}

Donde expX.s hace referencia al contenido de la variable expX y op.s hace referencia a la operación en cuestión.

- Condición compuesta. Una condición compuesta es aquella que tiene operaciones lógicas.

Si se enlazan condiciones mediante operadores lógicos AND y OR se emplea la técnica "del cortocircuito", de manera que si se enlazan cond1 y cond2 con un AND, (cond1 AND cond2), si cond1 es falso no se evalua cond2, dado que su función será falsa sea cual sea el valor de cond2.

Si el conector es OR, (cond1 OR cond2), cond2 sólo se evaluará si cond1 es falsa, pues en caso contrario, su disyunción será verdad para cualquier valor de cond2.

Cada vez que se reduce a una condición en base a expresiones, se genera el código de tres direcciones de la forma en que se ha explicado en el punto anterior.

En el momento en que se identifica un operador lógico, se sabe que a continuación se encontrará otra condición, que también generará un código con la misma estructura, así como otras dos etiquetas, una de certeza, y otra de falsedad. Ambas condiciones y el operador lógico, se reducirán a una condición. Ahora bien, dicha condición reducida deberá tener solo dos etiquetas, ¿qué etiquetas se le asignan?. La solución depende de la conectiva que se emplee.

- Conectiva AND.

Antes de la acción hay que tener el código asociado a las dos condiciones. Esto genera cuatro goto sin ninguna etq. Ahora tenemos que reducir los cuatro goto a dos, ya que una condición tiene que tener una etiqueta de verdad y otra de falsedad. Si se tiene cond1 AND cond2 el código que se tiene es el de la figura 4.6.5.5.

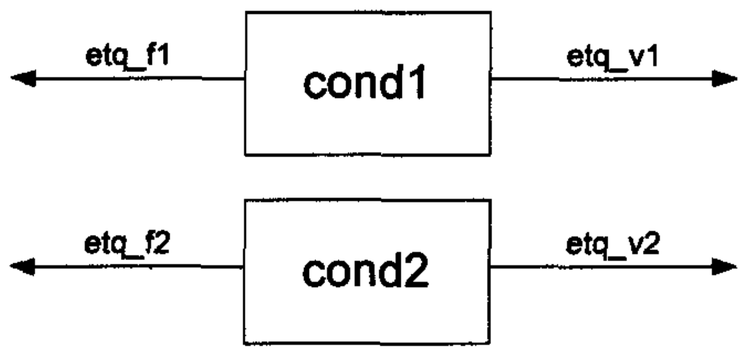


Figura 4.6.5.5. Dos condiciones.

El objetivo es obtener un bloque de código que represente entera la condición. Así que ese código tiene que tener una sola etq_verdad y una sola etq_falso, como se aprecia en la figura 4.6.5.6.

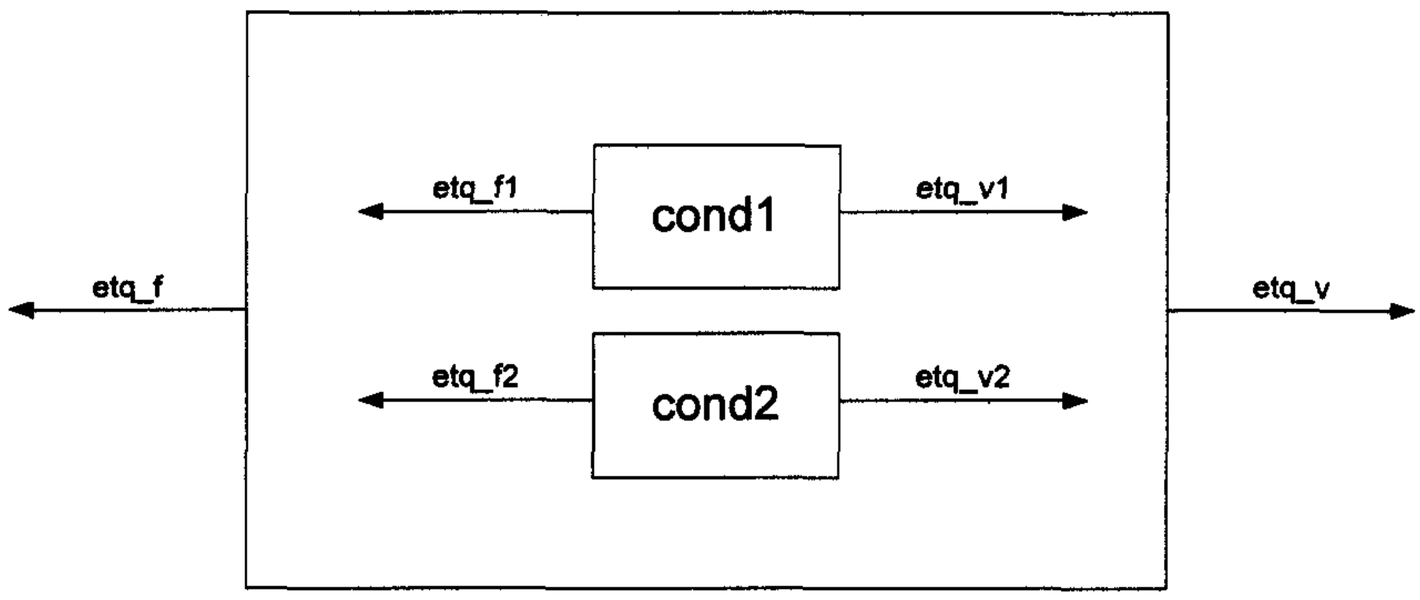


Figura 4.6.5.6. Encapsulamiento de dos condiciones.

Como es el caso de una conectiva AND, si cond1 es verdad, se tiene que evaluar cond2, como se observa en la figura 4.6.5.7.

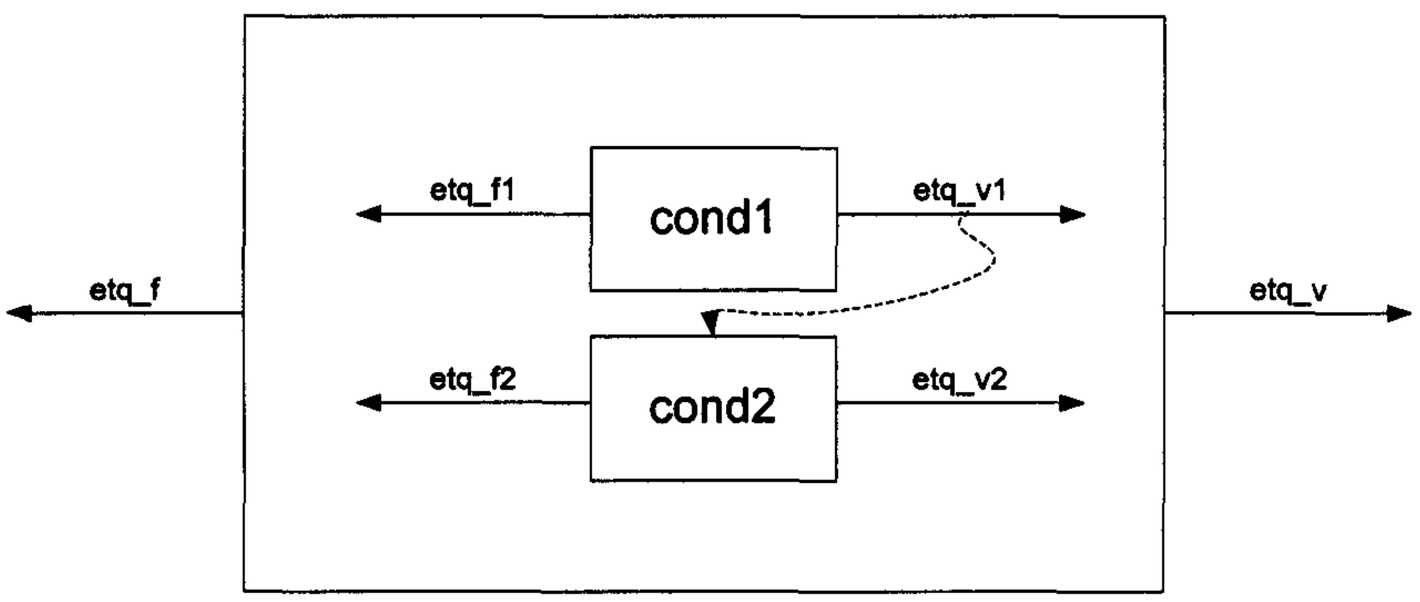


Figura 4.6.5.7. Salida de la cond1 verdadera.

Si cond1 es false hay que ir a la condición de falso de la cond2. ¿A donde irán las dos etiquetas de falso? A la condición de falso de todo el bloque. Figura 4.6.5.8.

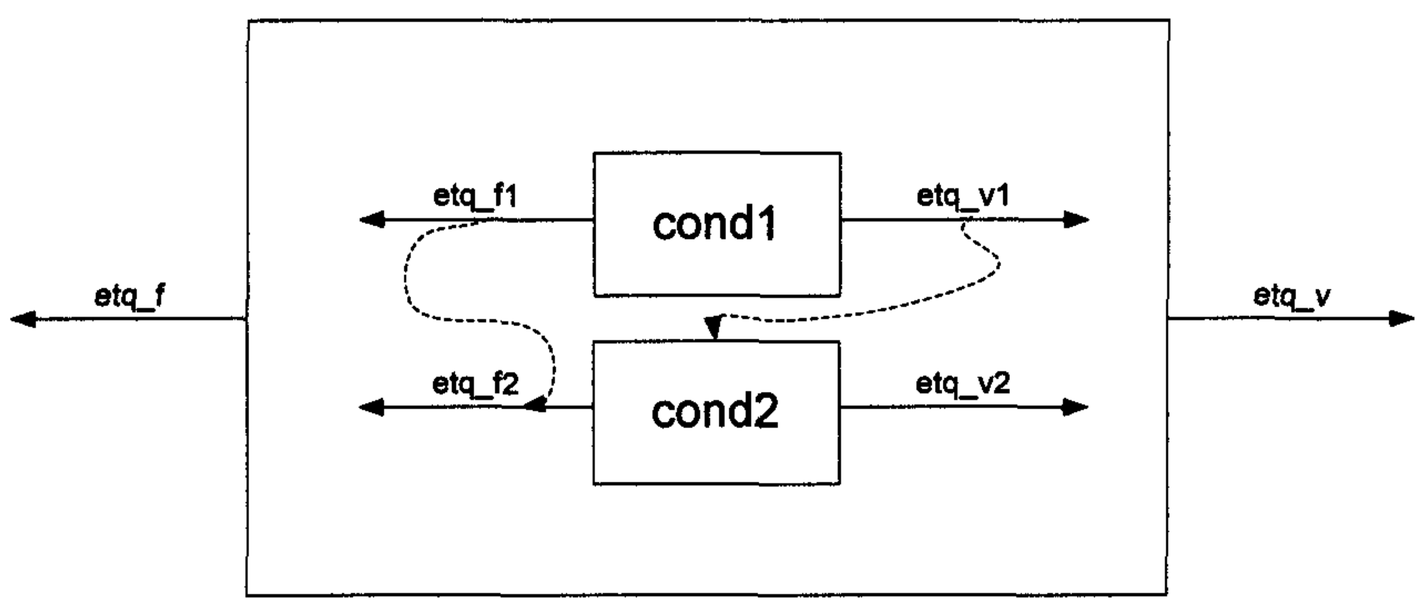


Figura 4.6.5.8. AND.

¿Cuales son las etiquetas de verdad y de falso de esta condición?

etq_V = etq_V2
etq_F = etq_F2
Por tanto, el código que se obtendría sería:

```
cond:  cond1 AND {escribe (cond1.etq_verdad);}
       cond2 {escribe (cond1.etq_falso);
              escribe (goto cond2.etq_falso);
              cond.etq_verdad := copia (cond2.etq_verdad);
              cond.etq_falso := copia (cond2.etq_falso);}
```

- Conectiva OR.

Es igual, pero invirtiendo los papeles de las etiquetas. Figura 4.6.5.9.

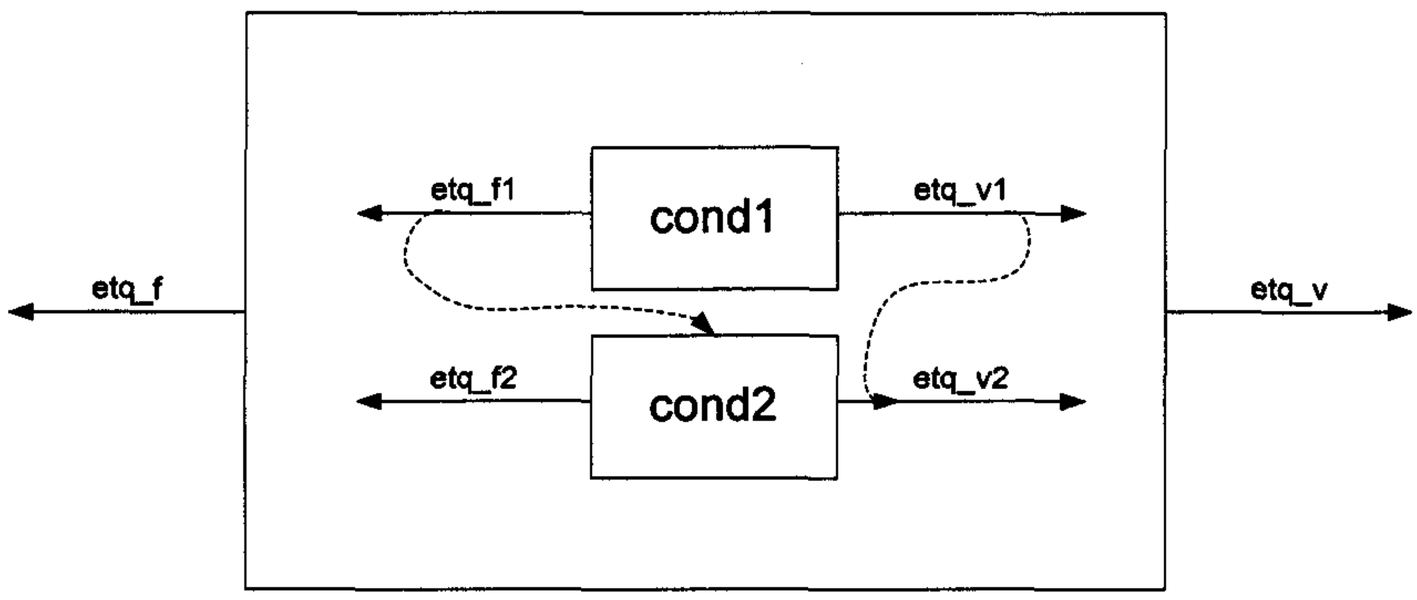


Figura 4.6.5.9. OR.

¿Cuales son las etiquetas de verdad y de falso de esta condición?

etq_V = etq_V2
etq_F = etq_F2
Por tanto, el código que se obtendría sería:

```
cond:  cond1 OR {escribe (cond1.etq_falso);}
       cond2 {escribe (cond1.etq_verdad);
              escribe (goto cond2.etq_verdad);
              cond.etq_verdad := copia (cond2.etq_verdad);
              cond.etq_falso := copia (cond2.etq_falso);}
```

- Conectiva NOT.

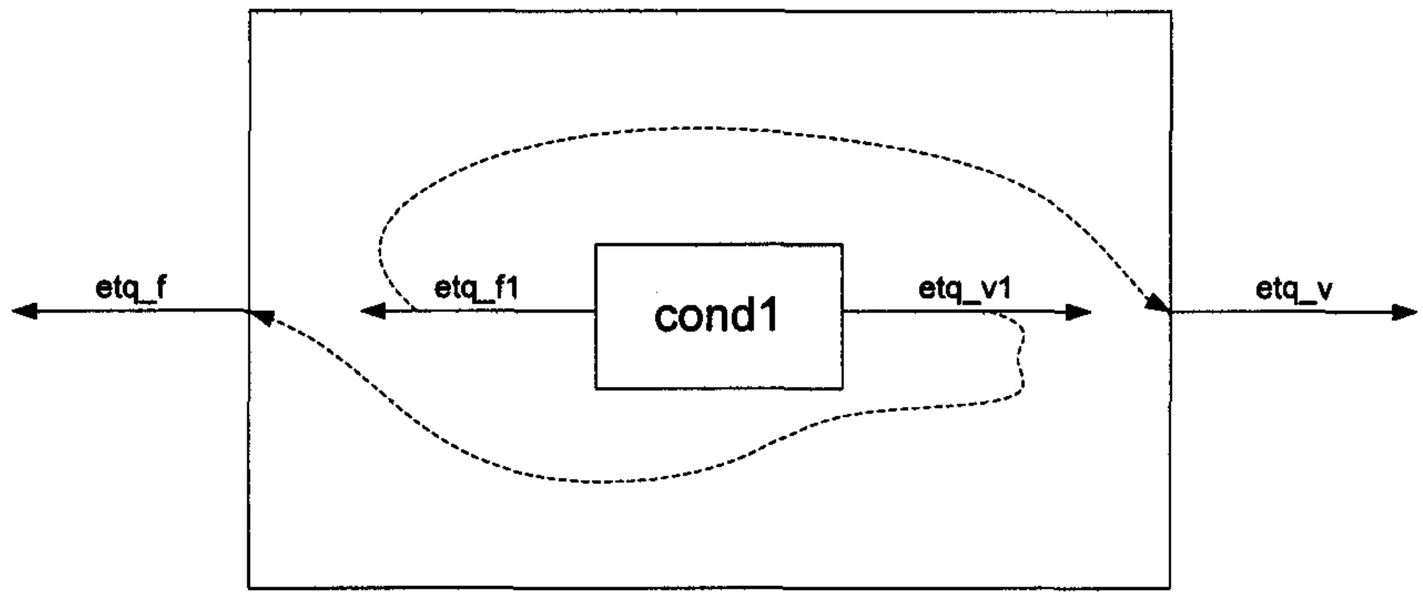


Figura 4.6.5.10. NOT.

El código asociado sería:

cond: NOT cond1 {cond.etq_verdad := copia (cond1.etq_falso);
cond.etq_falso := copia (cond1.etq_verdad);}

- Siguiendo este criterio podemos hacer cualquier operador lógico. Por ejemplo NAND, NOR,...

El caso del IF es el caso más simple. Basta, con indicar que la etiqueta de verdad de la condición está asociada al código a continuación del THEN, y la etiqueta de falso se asocia al código que puede haber tras el ELSE. En cualquier caso, una vez acabadas las sentencias del THEN se debe producir un salto al final del IF, porque no se tienen que ejecutar también las sentencias del ELSE. Por tanto, tras las sentencias del THEN, se crea una nueva etiqueta a la cual se saltará, y se coloca el destino de tal etiqueta al final del código del IF. Es decir, el código sería:

```
IF (IF.etq_fin := nueva_etiqueta()} cond THEN
        {escribe (cond.etq_verdad);}
        sent1
        {escribe (goto IF.etq_fin);}
ELSE
        {escribe (cond.etq_falso);}
        sent1
END_IF
{escribe (cond.etq_fin);}
```

4.6.5.4.2 Sentencia WHILE

El caso de WHILE y REPEAT es muy similar. En ambos casos se crea una etiqueta al comienzo del bucle, a la que se salta para repetir cada iteración.

En el caso del WHILE, a continuación se genera el código de la condición. La etiqueta de verdad se pone justo antes de las sentencias del WHILE, que es lo que se debe ejecutar si la condición es cierta. Al final de las sentencias se pondrá un salto al

inicio del bucle, donde de nuevo se comprobará la condición. La etiqueta de falso de la condición, se pondrá al final de todo lo relacionado con el WHILE, o lo que es lo mismo, al principio del código generado para las sentencias que siguen al WHILE.

WHILE {WHILE.etq_inicio := nueva_etiqueta();
escribe (WHILE.etq_inicio);}
cond DO {escribe (cond.etq_verdad);}
sent
END WHILE
{escribe (goto WHILE.etq_inicio);
escribe (cond.etq_falso);}

4.6.5.4.3 Sentencia REPEAT

Se crea una etiqueta al comienzo del bucle, a la que se salta para repetir cada iteración.

En el caso del REPEAT, a continuación de la etiqueta de comienzo del bucle se coloca el código de las sentencias, ya que la condición se evalúa al final. Tras las sentencias se coloca el código de la condición. Ahora, se debe hacer coincidir la etiqueta de comienzo del bucle con la etiqueta de falso de la condición. Como ambos nombres ya están asignados, la solución es colocar la etiqueta de falso, y en ella un goto al comienzo del bucle. Al final de todo el código asociado al REPEAT pondremos la etiqueta de verdad.

En este caso la sentencia se ejecuta siempre una vez, y después entra en la condición. Si la condición se cumple, se sale del bucle, y si no se hace una nueva iteración.

REPEAT {REPEAT.etq:= nueva_etiqueta();
escribe (REPEAT.etq);}
sent UNTIL cond {escribe (cond.etq_falso);
escribe (goto REPEAT.etq);
escribe (cond.etq_verdad);}

4.6.5.4.4 Sentencia CASE

La sentencia más compleja de todas es la sentencia CASE, ya que ésta permite un número indeterminado, y potencialmente infinito de condiciones, lo cual obliga a arrastrar una serie de parámetros a medida que se van efectuando reducciones.

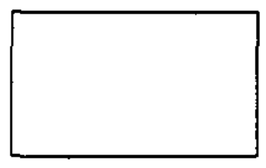
El problema es que la sentencia CASE necesita una recursión (no se puede utilizar una única regla de producción). Es necesario una regla de producción para la sentencia CASE diferente.

sent_case: inicio_case FIN CASE
| inicio_case OTHERWISE sent FIN case
;
inicio_case: CASE expr OF

| *inicio_Case CASO expr : sent*
;

Hay que considerar que aunque en cada caso aparezca sólo una expresión, se debe convertir en una comparación, tal que si es falsa se pase a comprobar la siguiente expresión, y si es cierta, se ejecute el código asociado, al final del cual se producirá un salto al final del CASE.

Cada uno de los casos va a generar un bloque de código diferente. Dentro de cada bloque todos los casos tienen la misma estructura. Ver figura 4.6.5.11.



Todos los bloques del CASE tienen que ser iguales. En todos ellos, se produce una comparación entre la expresión principal y la expresión subordinada.

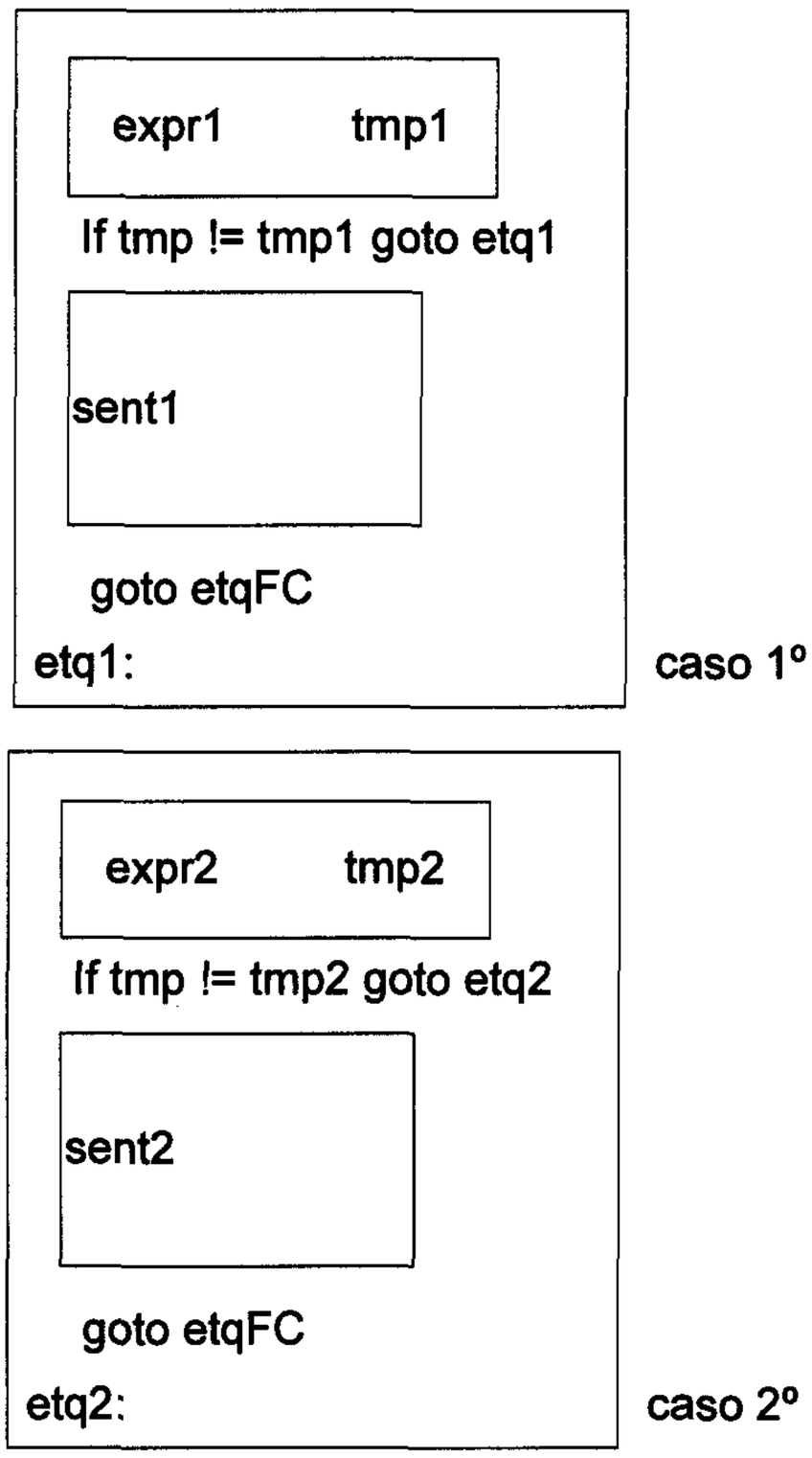


Figura 4.6.5.12. Comparaciones del CASE.

Por tanto, el CASE debe tener asociado los siguientes atributos:

- Una variable temporal que representa a la expresión que se va a ir comparando.
- Una etiqueta que representa el final del CASE y a la que se saltará tras poner el código asociado a la sentencia de cada CASO.

Así, el código asociado será:

```
sent_case: ic FIN CASE {excribe (ic.etq);}
        | ic OTHERWISE sent FIN CASE {escribe (ic.etq);}
        ;
ic : CASE expr OF {ic.var := expr.s;
                   ic.etq := nueva_etiqueta();}
   | ic1 CASO expr :
    {expr.etq := nueva_etiqueta();
     escribe (if ic.var != expr.s goto expr.etq);}
   sent {ic.var := copia (ic1.var);
         ic.etq := copia (ic1.etq);
         escribe (goto expr.etq);
         escribe (expr.etq);}
   ;
```

No es necesario colocar nada detrás del OTHERWISE porque después del último CASE, si no se cumple, se va al OTHERWISE.

4.6.5.5 Bloques básicos y diagramas de flujo

Los bloques básicos son trozos de código intermedio en los que el flujo de control es lineal, es decir, trozos de código cuyas instrucciones se ejecutan una detrás de otra sin saltos intermedios. Los bloques básicos tienen por tanto una única instrucción de comienzo de bloque y una única instrucción de salida del bloque.

Por ejemplo, la siguiente secuencia de instrucciones de tres direcciones forman un bloque básico.

```
(100) t1 := -c
(101) t2 := b * t1
(102) t3 := -c
(103) t4 := b * t3
(104) t5 := t2 * t4
(105) a := t5
(106) if (a>0) goto (712)
```

Para identificar los bloques básicos que componen una secuencia de instrucciones en código máquina puede seguirse el siguiente algoritmo:

1. Buscar los puntos de entrada, que corresponden a:
   a) La primera instrucción de código máquina.
   b) Cualquier instrucción situada en la dirección a la que se refiere una instrucción de salto condicional o incondicional.

c) Cualquier instrucción consecutiva a una instrucción de salto condicional o incondicional.

2. Para cada punto de entrada construir un bloque básico que vaya desde el punto de entrada hasta el siguiente punto de entrada.

3. Los arcos del diagrama de flujo se obtienen mediante las reglas:

a) Si el bloque básico termina en un salto incondicional, entonces incluir un arco hasta la dirección señalada.

b) Si el bloque básico termina en un salto condicional, entonces incluir un arco hasta la dirección señalada y otra al siguiente bloque básico.

Una instrucción de tres direcciones del tipo x:= y + z, se dice que define a x y que hace referencia a y (y también a z).

Una variable se dice que está activa en un determinado punto de la secuencia de instrucciones de tres direcciones si después de haber sido definida se hace referencia a ella en cualquier otro punto del programa que se ejecute posteriormente, (en el mismo bloque básico o en cualquier otro). Es decir, una variable está activa desde que nace con una definición, hasta que muere con el último uso que de ella hace cualquier instrucción del programa. Por el contrario, una variable esta inactiva, desde la última vez que se usa hasta que se define nuevamente.

El concepto de actividad e inactividad de las variables tiene importancia para la asignación de registros que se efectúa durante la generación de código máquina.

Ejemplo: Sea la función

```
void quicksort(int m, int n)
{
        int i,j,v,x;
        if (n<=m) return;
        i = m-1;
        j=n;
        v = a[n];
        while(1)
{
        i = i+1;
        while (v > a[i])
                j= j-1;
        while (a[j] > v)
                if (i>=j)
                        break;
}
```

```
        x = a[i];
        a[i] = a[n];
        a[j] = x;
        quicksort(m,j);
        quicksort(i+1,n);
}
```

El trozo de código fuente anterior da lugar a la siguiente secuencia de instrucciones de código intermedio.

```
(101) i := m - 1
(102) j := n
(103) t1 := n * 4
(104) v := a[t]
(105) i := i + 1
(106) t2 := 4 * i
(107) t3 := a[t2]
(108) if (t3 < v) goto (105)
(109) j := j - 1
(110) t4 := 4 * j
(111) t5 := a[t4]
(112) if (t5 > v) goto (109)
(113) if (i >= j) goto (23)
(114) t6 := 4 * i
(115) x := a[t6]
(116) t7 := 4 * i
(117) t8 := 4 * j
(118) t9 := a[t8]
(119) a[t7] := t9
(120) t10 := 4 * j
(121) a[t10] := x
(122) goto(105)
(123) t11 := 4 * i
(124) x := a[t11]
(125) t12 := 4 * i
(126) t13 := 4 * n
(127) t14 := a[t13]
(128) a[t12] := t14
(129) t15 := 4 * n
(130) a[t15] := x
```

A partir de éste se localizan los bloques básicos y se obtiene el diagrama de flujo que aparece a continuación en la figura 4.6.5.13:

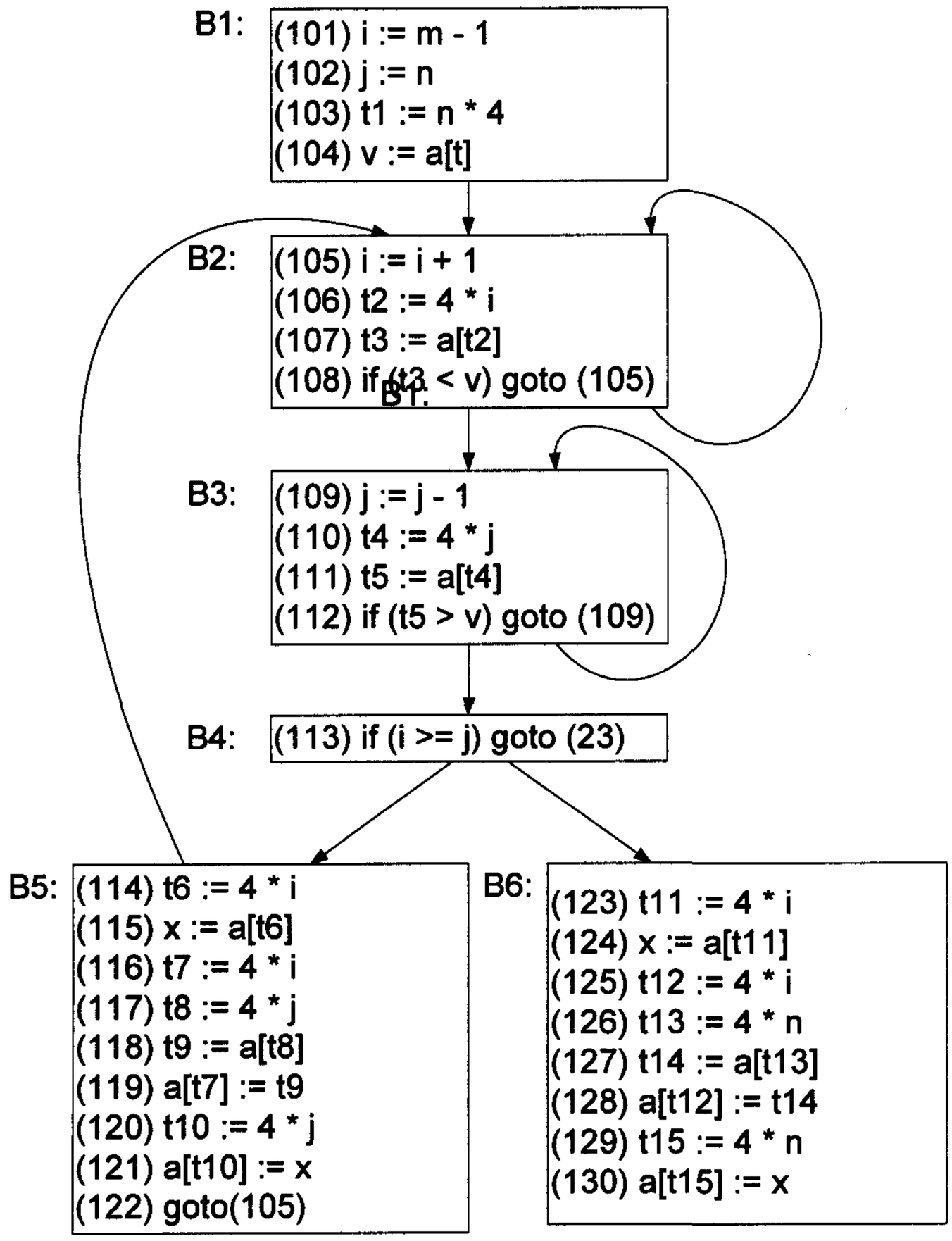


Figura 4.6.5.13. Diagrama de flujo.

4.6.6 Entorno de ejecución

4.6.6.1 Organización de la memoria en tiempo de ejecución

Cuando un programa se ejecuta sobre un Sistema Operativo existe un proceso previo llamado cargador que suministra al programa un bloque contiguo de memoria sobre el cual ha de ejecutarse. El programa resultante de la compilación debe organizarse de forma que haga uso de este bloque. Para ello el compilador incorpora al programa objeto el código necesario.

Las técnicas de gestión de la memoria durante la ejecución del programa difieren de unos lenguajes a otros, e incluso de unos compiladores a otros.

Para ello el diseño del mapa de memoria hay que determinar los siguientes tópicos:

- Permitir (o no) la recursividad en los procedimientos.
- ¿Qué se hace con los valores locales al salir de un procedimiento?
- Uso (o no) de valores no locales en los procedimientos.
- Técnica de paso de parámetros a los procedimientos.
- Pasar (o no) procedimientos como parámetros.
- Devolver (o no) procedimientos como resultados.
- Asignación (o no) dinámica de memoria bajo el control del programa.
- Se debe (o no) liberar la memoria después de usarla.

Para lenguajes imperativos, los compiladores generan programas que tendrán en tiempo de ejecución una organización de la memoria similar (a grandes rasgos) a la que aparece en la figura 4.6.6.1.

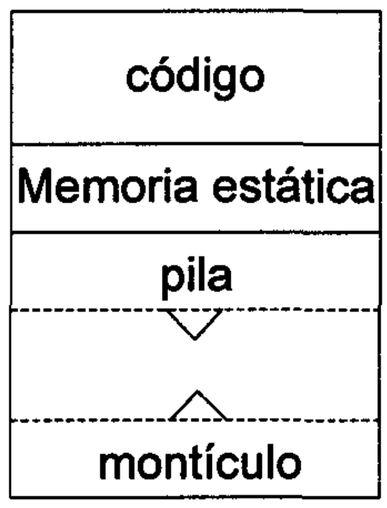


Figura 4.6.6.1. Organización de la memoria en ejecución.

En este esquema se distinguen las secciones de:

- Código.
- Memoria estática.
- Pila.
- Montículo.

4.6.6.2 El código

Es la zona donde se almacenan las instrucciones del programa ejecutable en código máquina, y también el código correspondiente a los procedimientos y funciones que utiliza. Su tamaño puede fijarse en tiempo de compilación.

Algunos compiladores fragmentan el código del programa objeto usando "overlays". Estos "overlays" son secciones de código objeto que se almacenan en ficheros independientes y que se cargan en la memoria central (RAM) dinámicamente, es decir, durante la ejecución del programa. Los ooverlays de un programa se agrupan en zonas y módulos, cada uno de los cuales contiene un conjunto de funciones o procedimientos.

Durante el tiempo de ejecución sólo uno de los módulos de cada uno de los overlays puede estar almacenado en memoria para cada overlay. El tamaño de esta zona debe ser igual al mayor módulo que se cargue sobre ella. Es función del programador determinar cuantas zonas de overlay se definen, qué funciones y procedimientos se encapsulan en cada módulo, y cómo se organizan estos módulos para ocupar cada uno de los overlays. Una restricción a tener en cuenta es que las funciones de un módulo no deben hacer referencia a funciones de otro módulo del mismo overlay, ya que nunca estarán simultáneamente en memoria. El siguiente ejemplo de la figura 4.6.6.2 se lo muestra gráficamente.

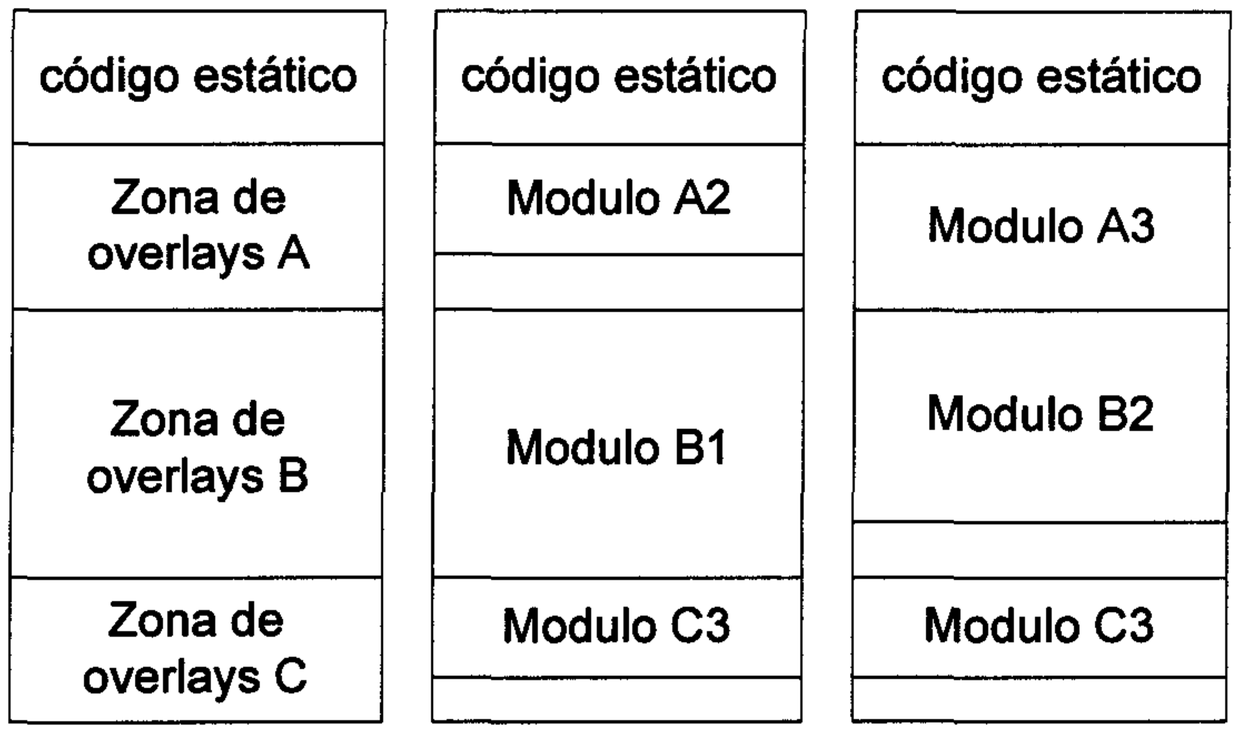


Figura 4.6.6.2. Distribución con overlays.

El tiempo de ejecución de un programa con overlays es mayor, puesto que durante la ejecución del programa es necesario cargar cada módulo cuando se realiza una llamada a alguna de las funciones que incluye. También es tarea del programador diseñar la estructura de overlays de manera que minimice el número de operaciones. La técnica de overlays se utiliza cuando el programa a compilar es muy grande en relación con la disponibilidad de memoria del sistema, o bien si se desea obtener programas de menor tamaño.

4.6.6.3 La memoria estática

La forma más fácil de almacenar el contenido de una variable en memoria en tiempo de ejecución es en memoria estática o permanente a lo largo de toda la ejecución del programa. No todos los objetos (variables) pueden ser almacenador estáticamente. Para que un objeto pueda ser almacenado en memoria estática su tamaño (número de bytes necesarios para su almacenamiento) ha de ser conocido en tiempo de compilación. Como consecuencia de esta condición no podrán almacenarse en memoria estática:

- Los objetos correspondientes a procedimientos o funciones recursivas, ya que en tiempo de compilación no se sabe el número de variables que serán necesarias.

- Las estructuras dinámicas de datos tales como listas, árboles,... ya que el número de elementos que la forman no es conocido hasta que el programa se ejecuta. Figura 4.6.6.3

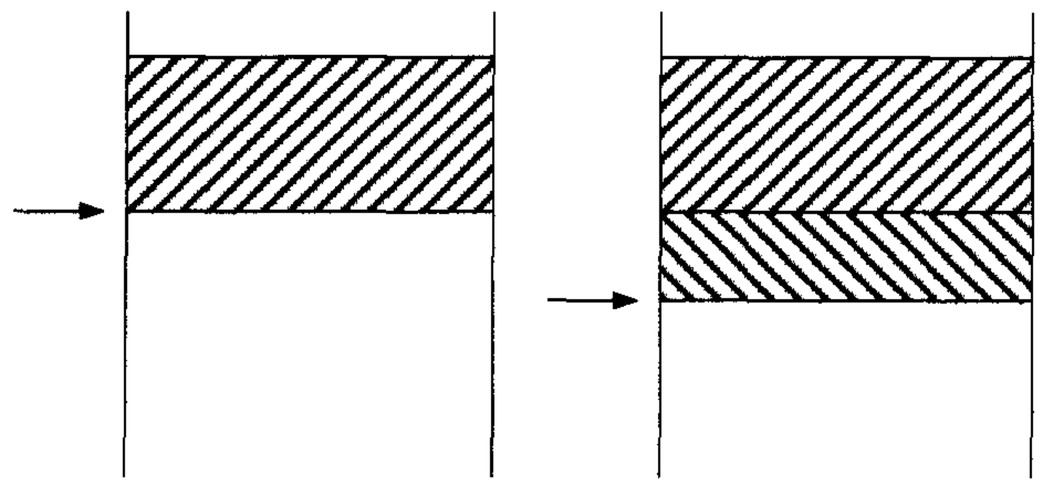


Figura 4.6.6.3. Alojamiento de un objeto.

Las técnicas de asignación de memoria estática son sencillas. A partir de una posición señalada por un puntero de referencia se aloja el objeto X, y se avanza el puntero tantos bytes como sean necesarios para almacenar el objeto X. La asignación de memoria puede hacerse en tiempo de compilación y los objetos están vigentes desde que comienza la ejecución del programa hasta que termina.

En los lenguajes que permiten la existencia de subprogramas, y siempre que todos los objetos de estos subprogramas puedan almacenarse estáticamente (por ejemplo en FORTRAN-IV) se aloja en la memoria estática un Registro de Activación correspondiente a cada uno de los subprogramas. Estos registros de activación

contendrán las variables locales, parámetros formales y valor devuelto por la función tal como indica la figura 4.6.6.5.

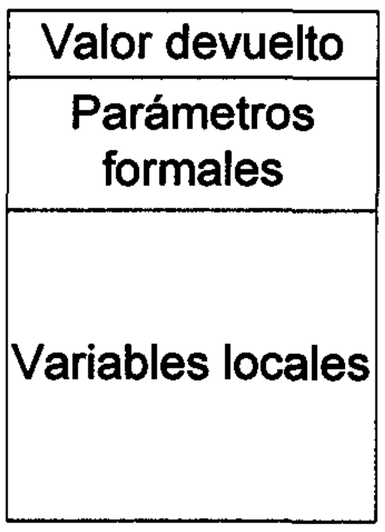


Figura 4.6.6.4. Registro de activación.

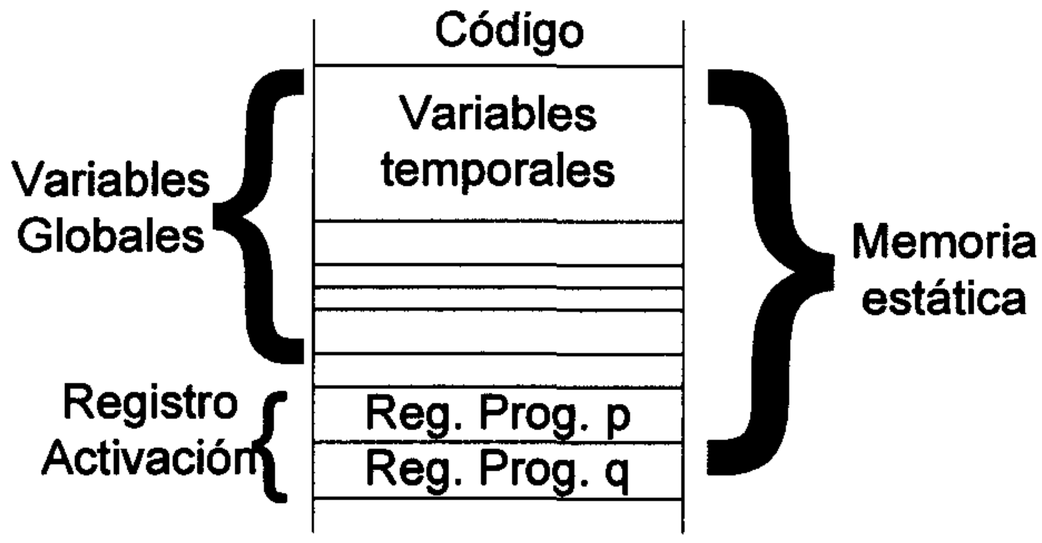


Figura 4.6.6.5. Estructura de la memoria estática en FORTRAN-IV.

Dentro de cada registro de activación las variables locales se organizan secuencialmente. Existe un sólo registro de activación para cada procedimiento y por tanto no están permitidas las llamadas recursivas. El proceso que se sigue cuando un procedimiento p llama a otro q es el siguiente:

1. p evalúa los parámetros de llamada, en caso de que se trate de expresiones complejas, usando para ello una zona de memoria temporal para el almacenamiento intermedio. Por ejemplo, si la llamada a q es q((3 *5) + (2 * 2),7) las operaciones previas a la llamada propiamente dicha en código máquina han de realizarse sobre alguna zona de memoria temporal. (En algún momento debe haber una zona de memoria que contenga el valor intermedio 15, y el valor intermedio 4 para sumarlos a continuación). En caso de utilización de memoria estática ésta zona de temporales puede ser común a todo el programa, ya que su tamaño puede deducirse en tiempo de compilación.

2. q inicializa sus variables y comienza su ejecución.

Dado que las variables están permanentemente en memoria es fácil implementar la propiedad de que conserven o no su contenido para cada nueva llamada.

4.6.6.4 La pila

La aparición de lenguajes con estructura de bloque trajo consigo la necesidad técnica de alojamiento en memoria más flexibles, que pudieran adaptarse a las demandas de memoria durante la ejecución del programa. En estos lenguajes, cada vez que comienza la ejecución de un procedimiento se crea un registro de activación para contener los objetos necesarios para su ejecución, eliminándolo una vez terminada ésta.

Dado que los bloques o procedimientos están organizados jerárquicamente, los distintos registros de activación asociados a cada bloque deberán colocarse en una pila en la que entrarán cuando comience la ejecución del bloque y saldrán al terminar el mismo (Figura 4.6.6.7). La estructura de los registros de activación varía de unos lenguajes a otros, e incluso de unos compiladores a otros. Este es uno de los problemas por los que a veces resulta difícil enlazar los códigos generados por dos compiladores diferentes. En general, los registros de activación de los procedimientos suelen tener algunos de los campos que pueden verse en la figura 4.6.6.6.

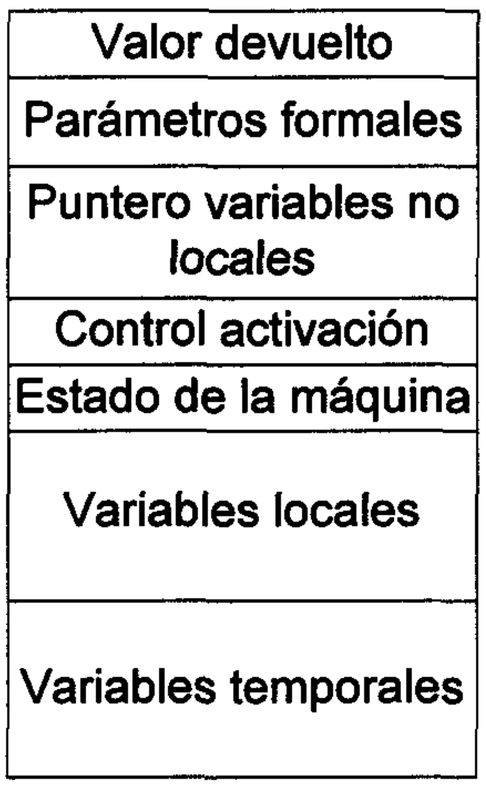


Figura 4.6.6.6. Registro de activación.

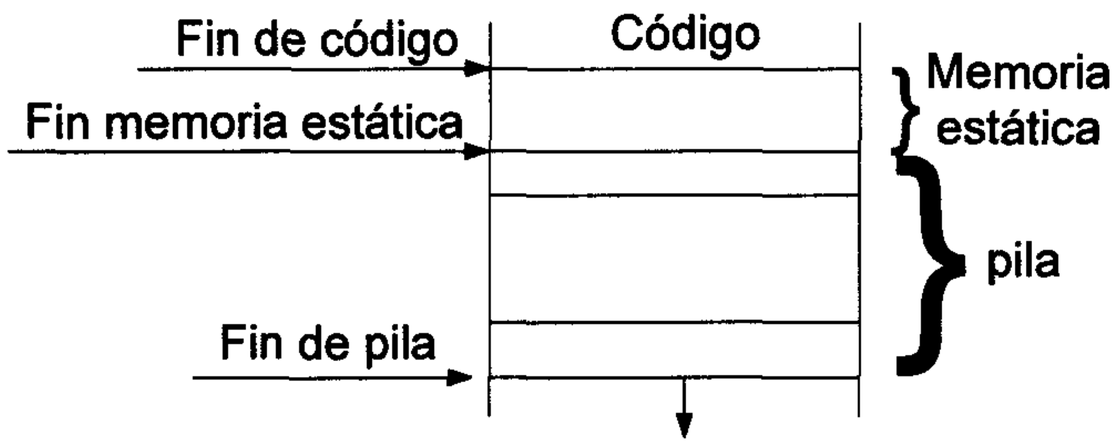


Figura 4.6.6.7. Estructura de la pila.

El puntero de control de activación guarda el valor que tenía el puntero de la cima de la pila antes de que entrase en ella el nuevo registro, de esta forma una vez que se desee desalojarlo puede restituirse e puntero de la pila a su posición original. Es

decir, es el puntero que se usa para la implementación de la estructura de datos "Pila" del compilador.

En la zona correspondiente al estado de la máquina se almacena el contenido que hubiera en los registros de la máquina antes de comenzar la ejecución del procedimiento. Estos valores deberán ser repuestos al finalizar la ejecución del procedimiento. El código encargado de realizar la copia del estado de la máquina es común para todos los procedimientos.

El puntero a las variables no locales permite el acceso a las variables declaradas en otros procedimientos. Normalmente no es necesario usar este campo puesto que puede conseguirse lo mismo con el puntero de control de activación, sólo tiene especial sentido cuando se utilizan procedimientos recursivos.

Al igual que en el alojamiento estático los registros de activación contendrán el espacio correspondiente a los parámetros formales (variables que aparecen en la cabecera) y las variables locales, (las que se definen dentro del bloque o procedimiento) así como una zona para almacenar el valor devuelto por la función y una zona de valores temporales para el cálculo de expresiones.

Para dos módulos o procedimientos diferentes, los registros de activación tendrán tamaños diferentes. Este tamaño por lo general es conocido en tiempo de compilación ya que se dispone de información suficiente sobre el tamaño de los objetos que lo componen. En ciertos casos esto no es así como por ejemplo ocurre en C cuando se utilizan arrays de dimensión indefinida. En estos casos el registro de activación debe incluir una zona de desbordamiento al final cuyo tamaño no se fija en tiempo de compilación sino solo cuando realmente llega a ejecutarse el procedimiento. Esto complica un poco la gestión de la memoria, por lo que algunos compiladores de bajo coste suprimen esta facilidad.

El procedimiento de gestión de la pila cuando un procedimiento q llama a otro procedimiento p, se desarrolla en dos fases, la primera de ellas corresponde al código que se incluye en el procedimiento q antes de transferir el control a p, y la segunda, al código que debe incluirse al principio de p para que se ejecute cuando reciba el control.

- El procedimiento autor de la llamada (q) evalúa las expresiones de la llamada, utilizando para ello su zona de variables temporales, y copia el resultado en la zona correspondiente a los parámetros formales del procedimiento que recibe la llamada.
- El autor de la llamada (q) coloca el puntero de control del procedimiento al que llama (p) de forma que apunte al final de la pila y transfiere el control al procedimiento al que llama (p).

- El receptor de la llamada (p) salva el estado de la máquina antes de comenzar su ejecución usando para ello la zona correspondiente de su registro de activación.

- El receptor de la llamada (p) inicializa sus variables y comienza su ejecución.

Al terminar la ejecución del procedimiento llamado (p) se desaloja su registro de activación procediendo también en dos fases. La primera se implementa mediante instrucciones al final del procedimiento que acaba de terminar su ejecución (p), y la segunda en el procedimiento que hizo la llamada tras recobrar el control:

- El procedimiento saliente (p) antes de finalizar su ejecución coloca el valor de retorno al principio de su registro de activación.

- Usando la información contenida en su registro el procedimiento que finaliza (p) restaura el estado de la máquina y coloca el puntero al final de pila en la posición en la que estaba originalmente.

- El procedimiento que realizó la llamada (q) copia el valor devuelto por el procedimiento al que llamó (p) dentro de su propio registro de activación (de q).

La figura 4.6.6.9 muestra el estado de la figura (durante el tiempo de ejecución) cuando se alcanza la instrucción señalada en el código de la figura 4.6.6.8.

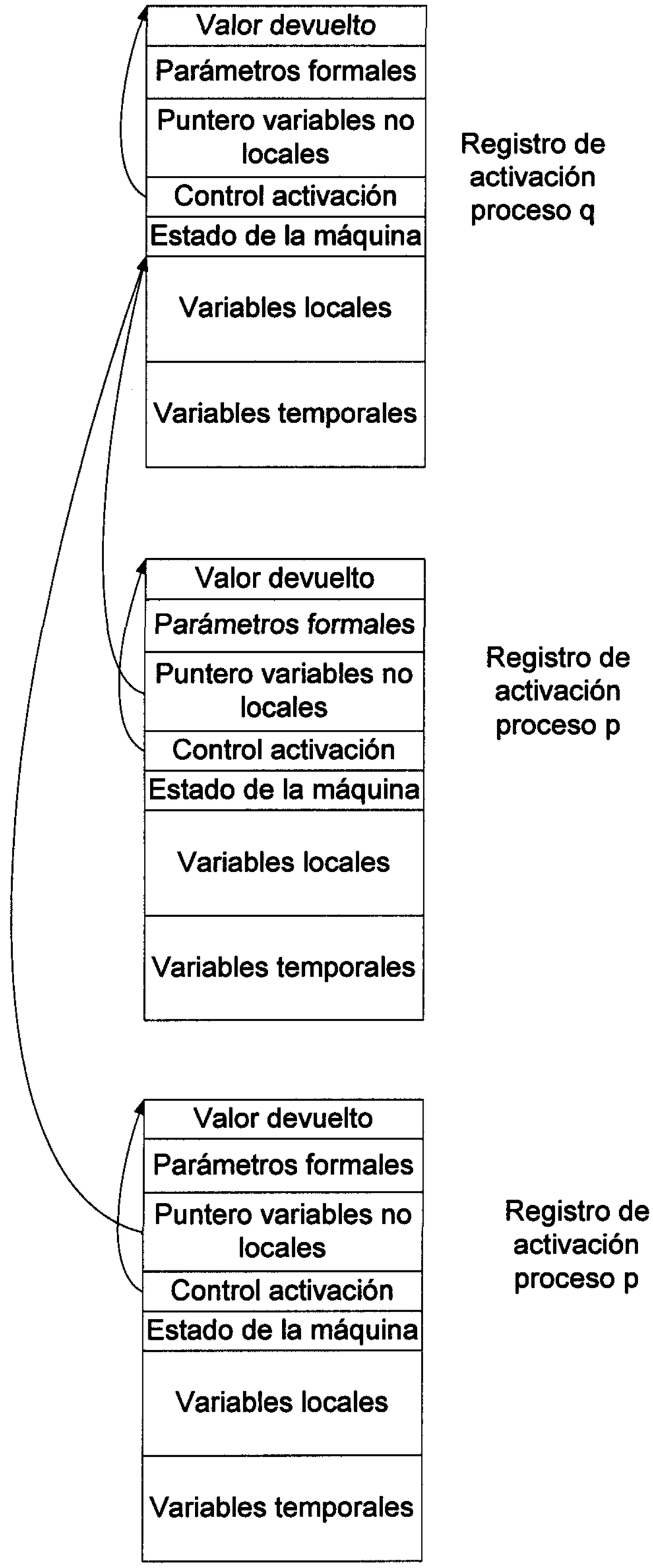


Figura 4.6.6.8. Registro de activación en la pila durante la ejecución del programa.

Dentro de un procedimiento, las variables locales se referencian siempre como direcciones relativas al comienzo del registro de activación, o bien al comienzo de la zona de variables locales. Por tanto, cuando sea necesario acceder desde un procedimiento a variables definidas de otros procedimientos cuyo ámbito sea accesible, será necesario proveer dinámicamente la dirección de comienzo de las variables de ese procedimiento. Para ello se utiliza dentro del registro de activación el puntero de enlace a variables no locales. Este puntero, señalará al comienzo de las variables locales del procedimiento inmediatamente superior. El puntero de enlace en una posición fija con respecto al comienzo de sus variables locales. Cuando los procedimientos se llaman a sí mismos recursivamente, el ámbito de las variables impide por lo general que una activación modifique las variables locales de otra activación del mismo procedimiento, por lo que en estos casos el procedimiento inmediato superior será, a efectos de enlace, el que originó la primera activación, tal como puede apreciarse en la figura 4.6.6.9.

```
funcion q(...// parámetros formales q)
{
   //var. locales de q
   .......
   X = p(23 +y*z....);
}
funcion p(...// parámetros formales p)
{
   //var. locales de p
   .......
   p(...);  <----------
   .......
}
```

Figura 4.6.6.9. Código fuente.

4.6.6.5 El montículo

Cuando el tamaño de un objeto a colocar en memoria puede variar en tiempo de ejecución, no es posible su ubicación en memoria estática, ni tan siquiera en la pila. Son ejemplos de este tipo de objetos las listas, los árboles, las cadenas de caracteres de longitud variable, etc... Para manejar este tipo de objetos el compilador debe disponer de un área de memoria de tamaño variable y que no se vea afectada por la activación o desactivación de procedimientos. Este trozo de memoria se llama montículo (heap). En aquellos lenguajes de alto nivel que requieran el uso del montículo, el compilador debe incorporar al programa objeto el código correspondiente a la gestión del montículo. Las operaciones básicas que se realizan sobre el montículo son:

- Alojamiento. Se demanda un bloque contiguo para poder almacenar un objeto de un cierto tamaño.

- Desalojo. Se indica que ya no es necesario conservar un objeto, y que por lo tanto, la memoria que ocupa debe quedar libre para ser reutilizada en caso necesario por otros objetos.

Según sea el programador o el propio sistema el que las invoque, estas operaciones pueden ser explícitas o implícitas respectivamente. En aso de alojamiento explícito el programador incluye en el código fuente una instrucción que demanda una cierta cantidad de memoria para la ubicación de un dato (por ejemplo en PASCAL mediante la instrucción new, en C mediante malloc). La cantidad de memoria requerida puede ser calculada por el compilador en función del tipo correspondiente al objeto que se desea alojar, o bien puede ser especificado directamente por el programador. El resultado de la función de alojamiento es por lo general un puntero a un trozo contiguo de memoria dentro del montículo que puede usarse para almacenar el valor del objeto. Los lenguajes de programación imperativos utilizan por lo general alojamiento y desalojo explícitos. Por el contrario los lenguajes lógicos y funcionales evitan al programador la tarea de manejar directamente los punteros realizando las operaciones implícitamente.

La gestión del montículo requiere técnicas adecuadas que optimicen el espacio que se ocupa o el tiempo de acceso, o bien ambos factores.

4.7 Requisitos

El requisito inicial era la continuación de un proyecto del proyecto de un simulador virtual de un laboratorio de ADN. Se requería unas modificaciones en el funcionamiento y añadir un módulo que permitiese introducir de forma textual unos comandos que eran las funciones de laboratorio.

La necesidad de los comandos era la introducción de protocolos ya definidos en teoría para la simulación de estos y así ahorrar el tiempo y gasto de su comprobación en la realidad.

Después al realizar una investigación más profunda de los protocolos que ya se habían definido, nos dimos cuenta de que la estructura era más compleja. Está estructura contenía bucles e incluso condiciones.

Viendo la evolución de los requisitos y la posibilidad de que estos cambiasen se decidió implementar la estructura más compleja pero que se podría adaptar más fácilmente a los cambios. Un compilador con todas sus fases con la pecularidad de que se iba a ejecutar en el propio programa.

4.8 Análisis

Tenía la necesidad y el desafío de definir un lenguaje que fuera sencillo, fácil de comprender y a golpe de vista entender. Está es la opción de diseño que más ha pesado, debido a que las personas que pueden programar en este lenguaje deberían ser profesionales en biología e informática, por eso debe ser muy intuitivo.

La base que tenemos son los protocolos que ya se han definido de forma teórica. Estos son sencillos, con una estructura de pseudos-código, como son intuitivos he decidido adaptarlos para que valgan como lenguaje en funciones. (página 21)

Como lo que se tiene que definir al fin y al cabo es un compilador también se decidió hacerlo como el lenguaje de programación C, debido a que ya se tenía avanzado por haber realizado un compilador de C++. Con esto ganamos las estructuras de las funciones y operaciones aritméticas.

4.8.1 Tecnología

Para la realización hemos utilizado Visual Studio .Net utilizando el lenguaje C#. Así aprendimos la tecnología .Net y queríamos utilizar un entorno visual fácil de aprender, de manejar y que nos sirviese para tener un mayor conocimiento de los lenguajes y tecnologías del mercado.

Un claro requisito era que el entorno fuese visual y con gran funcionalidad en esa parte. Por lo que también tubo peso para su elección debido a que Visual Studio con C# era un entorno visual sencillo de utilizar y comprender.

También tubo peso el lenguaje porque el lenguaje C es el que mayormente hemos utilizado en la carrera y es el que mayor dominamos de todos los existentes.

4.8.1.1 Requisitos mínimos.

El Visual Studio utilizado para la última versión del programa ha sido Visual Studio 2005 por lo que es necesario tener instalado en la máquina el Framework 2.0.

Después el programa no es muy exigente. El problema es especialmente la función Annealing, que conlleva una explosión combinatoria tan grande que es mejor tener un sistema con mucha memoria principal para que no acceda a disco en la ejecución. Aunque está es solo una recomendación.

4.8.2 Ciclo de vida

El ciclo de vida que elegimos al principio era el ciclo de vida en cascada pero en la práctica fue un ciclo de vida en espiral. Esto fue debido a que los requisitos que nos iban surgiendo fueron cambiando drásticamente y a la posterior decisión de realizar un

compilador. Así era necesario ir generando cada etapa del compilador y realizar las pruebas para determinar su correcta ejecución para continuar con el siguiente.

Más tarde en una entrevista de seguimiento nos dimos cuenta de que sería bueno tener funciones predefinidas en el sistema, como la función complementaria H, y alguna aleatoria, y eso hizo replantearse opciones de diseño que se tomaron.

4.8.3 Desarrollos posteriores

Viendo que los cambios en los requisitos podrían ser frecuentes. Se decidió asumir que la estructura que debíamos realizar fuese un compilador, la cual era la opción más compleja, pero la que podría asumir un impacto en los requisitos fuertes, debido a que este era más flexible que los interpretes que se proponían. (Interprete de sólo funciones o con alguna complejidad muy limitada).

Esto ha sido elegido porque se pretende que el proyecto no se estanque sino que sea desarrollado posteriormente con nuevas funciones de laboratorio que vayan saliendo, funciones predefinidas que debido a la complejidad que tenemos nos vengan muy bien…

4.9 Diseño

Para poder definir los analizadores lo mejor es irse a un ejemplo de código (Apartado 4.8.1).

4.9.1 Analizador léxico

Lo primero es identificar los tokens del lenguaje, definir cuales van a ser palabras reservadas. Para ello lo mejor es hacerse algunos ejemplos de lo que se quiere que sea el código y así determinar los tokens. Los tokens son:

(,), , , =, <, <=, >, >=, +, +=, ++, -, -=, --, *, *=, /, //, /=, %, %=, “, números (sólo enteros), IF, THEN, ELSE, END, FOR, TO, DO, WHILE, OR, NOT, AND, EQUAL, UNEQUAL, INCLUDE, FUNCTION, TRUE, FALSE, BOOLEAN, INTEGER, STRING, el resto son identificadores.

Se comentan las líneas precedidas por // hasta el final de la línea.

Los caracteres se definen con dobles comillas.

Todo esto se ha realizado en un módulo llamado léxico. Lo fundamental es la función siguienteToken, que recorre el texto y te devuelve el siguiente token encontrado.

4.9.2 Analizador sintáctico

El análisis que se ha utilizado a sido LL(1), ya que es el más sencillo de implementar, sin retroceso, y por la derecha. El único cuidado que hay que tener es que no empiecen por un mismo símbolo no terminal por el mismo símbolo terminal, y que en las λ, tampoco.

La gramática implementada es:

P → FUNCTION Pd P | id Pp P | (Lpfsalida) = id (Lpf) P | λ
*Pp → = Ek | += Ek | -= Ek | *= Ek | %= Ek | /= Ek | (Lpf) P*
Pd → (Ldfsalida) id (Ldf) C END | id Pd1
Pd1 → (Ldf) C END |id (Ldf) C END
C → IF Eb THEN C Oelse END C | FOR id = Ea TO Ea DO C END C |
WHILE Eb DO C END C | (Lpfsalida) = id (Lpf) C | id Cp C | λ
Oelse → ELSE C | λ
*Cp → = Ek | += Ek | -= Ek | *= Ek | %= Ek | /= Ek | ++ | -- | (Lpf) C*
Ea → Ek
Eb → Ek
Ek → E8
E8 → E7 E8p
E8p → OR E8 | λ
E7 → E6 E7p

E7p → AND E7 | λ
E6 → E5 E6p
E6p → EQUAL E6 | UNEQUAL E6 | λ
E5 → E4 E5p
E5p → > E5 | < E5 | <= E5 | >= E5 | λ
E4 → E3 E4p
E4p → + E4 | - E4 | λ
E3 → E2 E3p
*E3p → * E3 | / E3 | % E3 | λ*
E2 → NOT E1 | ++ E1 | -- E1 | + E1 | - E1 | E1 E2p
E2p → ++ | -- | λ
E1 → id Pf | (Ek) | numero | cadena | TRUE | FALSE
Pf → (Lpf) | λ
Lpf → Ek Lpfp | λ
Lpfp →, Ek Lpfp | λ
Ldfsalida → id Ldfsalidap
Ldfsalidap →,id Ldfsalidap| λ
Ldf → INTEGER id Ldfp | BOOLEAN id Ldfp | STRING id Ldfp | λ
Ldfp →, INTEGER id Ldfp | , BOOLEAN id Ldfp | , STRING id Ldfp | λ
Lpfsalida → id Lpfsalidap
Lpfsalidap →, Lpfsalida | λ

Como se puede apreciar P, es el axioma principal, va recogiendo información de declaraciones. El símbolo C, es el de código, y los E´s se refieren a las expresiones con operadores. Con los símbolos E´s garantizamos una preferencia de operadores, la cual es heredada del lenguaje C.

4.9.3 Analizador semántico

Las acciones semánticas son las siguientes:

P → FUNCTION Pd{P.error := Pd.error;} P_1 {P.error:=P_1.error;}
P → id Pp{P.error := Pp.error; Pp := ObtenerId(id);} P_1 {P.error:=P_1.error;}
P → (Lpfsalida) = id (Lpf) {if (ObtenerId(id).estructura = funcion /\
Lpf.num_parametros = ObtenerId(id).parametros.num_parametros)
For (i=0; i< Lpf.num_parametros; i++)
if (Lpf[i].tipo ¡= ObtenerId(id).parametros[i].tipo)
P.error := true;
if (ObtenerId(id).tipo = funcion /\ Lpfsalida.num_parametros =
ObtenerId(id).valor.num_parametros)
For (i=0; i< Lpfsalida.num_parametros; i++)
if (Lpfsalida[i].estructura = identificador)
Lpfsalida[i].tipo := ObtenerId(id).valor[i].tipo;
else
P.error := true;

if (ObtenerId(id).tipo = entero ∨ ObtenerId(id).tipo = booleano ∨
ObtenerId(id).tipo = cadena)
Lpfsalida.num_parametros = 1;
Lpfsalida.estructura = identificador;
else
P.error := true;
Lpfsalida.tipo := id.tipo;
} P_1 {P.error:=P_1.error;}
P → λ
Pp → = *Ek*{ if (Ek.es_funcion_laboratorio = false ∨ Pp.estructura = funcion ∨ (
Ek.tipo ¡= cadena ∧ Ek.tipo ¡= entero ∧ Ek.tipo ¡= booleano) ∨ (Pp.tipo ¡=
ninguno ∧ Ek.tipo ¡= Pp.tipo))
Pp.error := true;
else
Pp.error := Ek.error;
Pp.estructura := identificador;
Pp.tipo := Ek.tipo;}
Pp → += *Ek*{ if (Ek.es_funcion_laboratorio = false ∨ Pp.estructura = funcion ∨
(Pp.tipo ¡= entero ∧ Ek.tipo ¡= entero) ∨ (Pp.tipo ¡= cadena ∨ (Ek.tipo ¡= cadena
∧ Ek.tipo ¡= entero))
Pp.error := true;
else
Pp.error := Ek.error;}
Pp → -= { if (Ek.es_funcion_laboratorio = false ∨ Pp.estructura = funcion ∨
Ek.tipo ¡= entero ∨ Pp.tipo ¡= entero)
Pp.error := true;
else
Pp.error := Ek.error;}
Pp → *= *Ek* { if (Ek.es_funcion_laboratorio = false ∨ Pp.estructura = funcion ∨
Ek.tipo ¡= entero ∨ Pp.tipo ¡= entero)
Pp.error := true;
else
Pp.error := Ek.error;}
Pp → %= *Ek* { if (Ek.es_funcion_laboratorio = false ∨ Pp.estructura = funcion ∨
Ek.tipo ¡= entero ∨ Pp.tipo ¡= entero)
Pp.error := true;
else
Pp.error := Ek.error;}
Pp → /= *Ek* { if (Ek.es_funcion_laboratorio = false ∨ Pp.estructura = funcion ∨
Ek.tipo ¡= entero ∨ Pp.tipo ¡= entero)
Pp.error := true;
else
Pp.error := Ek.error;}
Pp → *(Lpf)* {
if (Pp.estructura = funcion ∧ Pp.num_parametros = Lpf.num_parametros)
For (i=0; i< Lpf.num_parametros; i++)

if (Lpf[i].tipo ¡= Pp.parametros[i].tipo)
Pp.error := true;
}*P*{ Pp.error := P.error; }
Pd → *(Ldfsalida) id {*
if(ObtenerId(id).estructura ¡=funcion /\ ObtenerId(id).estructura ¡=funcion_laboratorio)
InsertarId(id).estructura := funcion;
InsertarId(id).tipo := funcion;
InsertarId(id).valor:= Ldfsalida.parametros;
InsertarId(id).num_parametros := Ldf.num_parametros;
else
Pd.error := true;
} (Ldf) { InsertarId(id).parametros := Ldf.parametros;
InsertarId(id).parametros.num_parametros := Ldf.num_parametros;}
C{Pd.error:= C.error;} *END*
Pd → *id Pd1* { Pd.error := Pd1.error;
Pd.estructura := Pd1.estructura;
Pd.tipo := Pd1.tipo;
Pd.valor := Pd1.valor;
Pd.num_parametros := Pd1.num_parametros;
Pd.parametros := Pd1.parametros;
Pd.parametros.num_parametros := Pd1.parametros.num_parametros;
}
Pd1 → *(*
{ if not (NoEstaEnTSGlobal(Pd1.nombre) ObtenerId(Pd1).estructura ¡= funcion /\ ObtenerId(Pd1).estructura ¡= funcion_laboratorio /\ ObtenerId(Pd1).estructura := funcion)
Pd1.error := true;}
Ldf { Pd1.error:= Ldf.error;
InsertarId(Pd1).num_parametros := Ldf.num_parametros;
InsertarId(Pd1).parametros := Ldf.parametros;}
) C {Pd1.error:= C.error;} *END*
Pd1 → *id (*
{ if not(NoEstaEnTSGlobal(id.nombre) /\ ObtenerId(id).estructura ¡= funcion /\ ObtenerId(id).estructura ¡= funcion_laboratorio /\ ObtenerId(id).estructura := funcion)
Pd1.error := true;}
Ldf { Pd1.error:= Ldf.error;
InsertarId(id).num_parametros := Ldf.num_parametros;
InsertarId(id).parametros := Ldf.parametros;}
) C {Pd1.error:= C.error;} *END*
C → *IF Eb* { C.error := Eb.error;} *THEN* C_1 { C.error := C_1.error;}*Oelse*{ C.error := Oelse.error;} *END* C_2 { C.error := C_2.error; }
C → *FOR id = Ea* {
if (ObtenerId(id).estructura != function)
InsertarId(id).estructura := identificador;
InsertarId(id).tipo := Ea.tipo;

else
C.error := true;
} *TO* Ea_1 { C.error := Ea_1.error;} *DO C1* { C.error := C_1.error;} *END C2*{
C.error := C_2.error;}
C → *WHILE Eb* {C.error := Eb.error; } *DO* C_1 {C.error := C_1.error; } *END* C_2 {C.error := C_2.error; }
C → *(Lpfsalida) = id (Lpf)*
{if (ObtenerId(id).estructura = funcion /\ Lpf.num_parametros = ObtenerId(id).parametros.num_parametros)
For (i=0; i< Lpf.num_parametros; i++)
if (Lpf[i].tipo ¡= ObtenerId(id).parametros[i].tipo)
C.error := true;
if (ObtenerId(id).tipo = funcion /\ Lpfsalida.num_parametros = ObtenerId(id).valor.num_parametros)
For (i=0; i< Lpfsalida.num_parametros; i++)
if (Lpfsalida[i].estructura = identificador)
Lpfsalida[i].tipo := ObtenerId(id).valor[i].tipo;
else
C.error := true;
if (ObtenerId(id).tipo = entero \/ ObtenerId(id).tipo = booleano \/ ObtenerId(id).tipo = cadena)
Lpfsalida.num_parametros = 1;
Lpfsalida.estructura = identificador;
else
C.error := true;
Lpfsalida.tipo := id.tipo;
else if (ObtenerId(id).estructura = funcion_laboratorio)
if (ObtenerId(id).nombre =Mix /\ Lpf.num_parametros.Count > 1)
For (i=0; i< Lpf.num_parametros; i++)
if (Lpf[i].tipo ¡= cadena)
C.error := true;
else if (ObtenerId(id).nombre =Delete /\ Lpf.num_parametros.Count > 0)
For (i=0; i< Lpf.num_parametros; i++)
if (Lpf[i].tipo ¡= cadena)
C.error := true;
else if (Lpf.num_parametros = id.parametros.num_parametros)
For (i=0; i< Lpf.num_parametros; i++)
if (Lpf[i].tipo ¡= ObtenerId(id).parametros[i].tipo)
C.error := true;
If (ObtenerId(id).tipo = funcion /\ Lpfsalida.num_parametros = ObtenerId(id).valor.num_parametros)
For (i=0; i< Lpfsalida.num_parametros; i++)
if (Lpfsalida[i].estructura = identificador)
Lpfsalida[i].tipo := ObtenerId(id).valor[i].tipo;
else
C.error := true;

if (ObtenerId(id).tipo = entero ∨ ObtenerId(id).tipo = booleano ∨ ObtenerId(id).tipo = cadena)
If (Lpfsalida.num_parametros ¡= 1 ∨ Lpfsalida.estructura ¡= identificador)
C.error := true;
else if (ObtenerId(id).tipo ¡= funcion ∨ Lpfsalida.num_parametros ¡= 0)
C.error := true;
else
C.error := true; } C_1 { C.error := C_1.error; }
C → *id* {
Cp.estructura := ObtenerId(id).estructura;
Cp. tipo:= ObtenerId(id). tipo;
} *Cp* {C.error:= Cp.error; } C_1 { C.error := C_1.error;}
C → *λ* {C.error := false; }
Oelse → *ELSE C* {Oelse.error := C.error; }
Oelse → *λ* {Oelse.error := false; }
Cp → = *Ek* { if (Ek.es_funcion_laboratorio = false ∨ Cp.estructura = funcion ∨ (Ek.tipo ¡= cadena ∧ Ek.tipo ¡= entero ∧ Ek.tipo ¡= booleano) ∨ (Cp.tipo ¡= ninguno ∧ Ek.tipo ¡= Cp.tipo))
Cp.error := true;
else
Cp.error := Ek.error;
Cp.estructura := identificador;
Cp.tipo := Ek.tipo;}
Cp → += *Ek* { if (Ek.es_funcion_laboratorio = false ∨ Cp.estructura = funcion ∨ (Cp.tipo ¡= entero ∧ Ek.tipo ¡= entero) ∨ (Cp.tipo ¡= cadena ∨ (Ek.tipo ¡= cadena ∧ Ek.tipo ¡= entero))
Cp.error := true;
else
Cp.error := Ek.error;}
Cp → -= *Ek*{ if (Ek.es_funcion_laboratorio = false ∨ Cp.estructura = funcion ∨ Ek.tipo ¡= entero ∨ Cp.tipo ¡= entero)
Cp.error := true;
else
Cp.error := Ek.error;}
Cp → *= *Ek* { if (Ek.es_funcion_laboratorio = false ∨ Cp.estructura = funcion ∨ Ek.tipo ¡= entero ∨ Cp.tipo ¡= entero)
Cp.error := true;
else
Cp.error := Ek.error;}
Cp → %= *Ek* { if (Ek.es_funcion_laboratorio = false ∨ Cp.estructura = funcion ∨ Ek.tipo ¡= entero ∨ Cp.tipo ¡= entero)
Cp.error := true;
else
Cp.error := Ek.error;}

Cp → /= *Ek*{ if (Ek.es_funcion_laboratorio = false ∨ Cp.estructura = funcion ∨ Ek.tipo ¡= entero ∨ Cp.tipo ¡= entero)
Cp.error := true;
else
Cp.error := Ek.error;}
Cp → ++ {if (Cp.estructura != identificador ∨ Cp.tipo!=entero)
Cp.error := true;
else
Cp.error := false;}
Cp → -- {if (Cp.estructura != identificador ∨ Cp.tipo!=entero)
Cp.error := true;
else
Cp.error := false;}
Cp → *(Lpf)* {
if (Cp.estructura = funcion /\ Cp.num_parametros = Lpf.num_parametros)
For (i=0; i< Lpf.num_parametros; i++)
if (Lpf[i].tipo ¡= Cp.parametros[i].tipo)
Cp.error := true;
else if (Cp.estructura = funcion_laboratorio)
if (Cp.nombre =Delete /\ Lpf.num_parametros.Count > 0)
For (i=0; i< Lpf.num_parametros; i++)
if (Lpf[i].tipo ¡= cadena)
Cp.error := true;
else if (Cp.nombre =AddChaynADN /\ Lpf.num_parametros.Count = 2)
For (i=0; i< Lpf.num_parametros; i++)
if (Lpf[i].tipo ¡= cadena)
Cp.error := true;
else if (Cp.nombre =AddADN /\ Lpf.num_parametros.Count = 3)
For (i=0; i< Lpf.num_parametros; i++)
if (Lpf[i].tipo ¡= cadena)
Cp.error := true;
else
Cp.error := true; }*C*{ Cp.error := C.error; }
Ea → *Ek*{ Ea.es_funcion_laboratorio := Ek.es_funcion_laboratorio;
Ea.error := Ek.error;
if (Ek.es_funcion_laboratorio = false /\ Ek.tipo = entero)
Ea.estructura := Ek.estructura;
Ea.tipo := Ek.tipo;
else
Ea.error := true;}
Eb → *Ek*{ Eb.es_funcion_laboratorio := Ek.es_funcion_laboratorio;
Eb.error := Ek.error;
if (Ek.es_funcion_laboratorio = false /\ Ek.tipo = booleano)
Eb.estructura := Ek.estructura;
Eb.tipo := Ek.tipo;
else

Eb.error := true;}
Ek → { Ek.es_funcion_laboratorio := false;} *E8* {
Ek.es_funcion_laboratorio := E8.es_funcion_laboratorio;
Ek.error := E8.error;
Ek.estructura := E8.estructura;
Ek.tipo := E8.tipo;}
E8 → *E7* {E8.es_funcion_laboratorio := E7.es_funcion_laboratorio;
E8.error := E7.error;
E8.estructura := E7.estructura;
E8.tipo := E7.tipo;
E8p.estructura := E7.estructura;
E8p.tipo := E7.tipo;
} E8p{
if (E8.es_funcion_laboratorio ∨ E8.error)
E8.es_funcion_laboratorio := E8p.es_funcion_laboratorio;
E8.error := E8p.error;
E8.estructura := E8p.estructura;
E8.tipo := E8p.tipo; }
E8p → *OR* { E8.es_funcion_laboratorio := false;
E8.estructura := E8p.estructura;
E8.tipo := E8p.tipo;
} *E8* {if (E8.es_funcion_laboratorio = false ∧ E8.tipo = E8p.tipo ∧ E8.tipo != booleano)
E8p.es_funcion_laboratorio := E8.es_funcion_laboratorio;
E8p.error := E8.error;
E8p.estructura := temporal;
E8p.tipo := E8.tipo;
else
E8p.error := true;}
E8p → *λ*{ E8p.error := false; E8p.es_funcion_laboratorio:= false;}
E7 → *E6*{ E7.es_funcion_laboratorio := E6.es_funcion_laboratorio;
E7.error := E6.error;
E7.estructura := E6.estructura;
E7.tipo := E6.tipo;
E7p.estructura := E6.estructura;
E7p.tipo := E6.tipo;
} E7p{
if (E7.es_funcion_laboratorio ∨ E7.error)
E7.es_funcion_laboratorio := E7p.es_funcion_laboratorio;
E7.error := E7p.error;
E7.estructura := E7p.estructura;
E7.tipo := E7p.tipo; }
E7p → *AND* { E7.es_funcion_laboratorio := false;
E7.estructura := E7p.estructura;
E7.tipo := E7p.tipo;

} *E7* {if (E7.es_funcion_laboratorio = false /\ E7.tipo = E7p.tipo /\ E7.tipo != booleano)
E7p.es_funcion_laboratorio := E7.es_funcion_laboratorio;
E7p.error := E7.error;
E7p.estructura := temporal;
E7p.tipo := E7.tipo;
else
E7p.error := true;}
E7p → *λ*{ E7p.error := false; E7p.es_funcion_laboratorio:= false;}
E6 → *E5*{ E6.es_funcion_laboratorio := E5.es_funcion_laboratorio;
E6.error := E5.error;
E6.estructura := E5.estructura;
E6.tipo := E5.tipo;
E6p.estructura := E5.estructura;
E6p.tipo := E5.tipo;
} E6p{
if (E6.es_funcion_laboratorio \/ E6.error)
E6.es_funcion_laboratorio := E6p.es_funcion_laboratorio;
E6.error := E6p.error;
E6.estructura := E6p.estructura;
E6.tipo := E6p.tipo; }
E6p → *EQUAL* { E6.es_funcion_laboratorio := false;
E6.estructura := E6p.estructura;
E6.tipo := E6p.tipo;
} *E6* {if (E6.es_funcion_laboratorio = false /\ E6.tipo = E6p.tipo /\ E6.tipo != ninguno)
E6p.es_funcion_laboratorio := E6.es_funcion_laboratorio;
E6p.error := E6.error;
E6p.estructura := temporal;
E6p.tipo := E6.tipo;
else
E6p.error := true;}
E6p → *UNEQUAL* { E6.es_funcion_laboratorio := false;
E6.estructura := E6p.estructura;
E6.tipo := E6p.tipo;
} *E6* {if (E6.es_funcion_laboratorio = false /\ E6.tipo = E6p.tipo /\ E6.tipo != ninguno)
E6p.es_funcion_laboratorio := E6.es_funcion_laboratorio;
E6p.error := E6.error;
E6p.estructura := temporal;
E6p.tipo := E6.tipo;
else
E6p.error := true;}
E6p → *λ*{ E6p.error := false; E6p.es_funcion_laboratorio:= false;}
E5 → *E4*{ E5.es_funcion_laboratorio := E4.es_funcion_laboratorio;
E5.error := E4.error;

E5.estructura := E4.estructura;
E5.tipo := E4.tipo;
E5p.estructura := E4.estructura;
E5p.tipo := E4.tipo;
} E5p{
if (E5.es_funcion_laboratorio ∨ E5.error)
E5.es_funcion_laboratorio := E5p.es_funcion_laboratorio;
E5.error := E5p.error;
E5.estructura := E5p.estructura;
E5.tipo := E5p.tipo; }
E5p → >{ E5.es_funcion_laboratorio := false;
E5.estructura := E5p.estructura;
E5.tipo := E5p.tipo;
} *E5* {if (E5.es_funcion_laboratorio = false ∧ E5.tipo = entero ∧ E5p.tipo = entero)
E5p.es_funcion_laboratorio := E5.es_funcion_laboratorio;
E5p.error := E5.error;
E5p.estructura := temporal;
E5p.tipo := E5.tipo;
else
E5p.error := true;}
E5p → <{ E5.es_funcion_laboratorio := false;
E5.estructura := E5p.estructura;
E5.tipo := E5p.tipo;
} *E5* {if (E5.es_funcion_laboratorio = false ∧ E5.tipo = entero ∧ E5p.tipo = entero)
E5p.es_funcion_laboratorio := E5.es_funcion_laboratorio;
E5p.error := E5.error;
E5p.estructura := temporal;
E5p.tipo := E5.tipo;
else
E5p.error := true;}
E5p → <= { E5.es_funcion_laboratorio := false;
E5.estructura := E5p.estructura;
E5.tipo := E5p.tipo;
} *E5* {if (E5.es_funcion_laboratorio = false ∧ E5.tipo = entero ∧ E5p.tipo = entero)
E5p.es_funcion_laboratorio := E5.es_funcion_laboratorio;
E5p.error := E5.error;
E5p.estructura := temporal;
E5p.tipo := E5.tipo;
else
E5p.error := true;}
E5p → >={ E5.es_funcion_laboratorio := false;
E5.estructura := E5p.estructura;
E5.tipo := E5p.tipo;

} *E5* {if (E5.es_funcion_laboratorio = false /\ E5.tipo = entero /\ E5p.tipo = entero)
E5p.es_funcion_laboratorio := E5.es_funcion_laboratorio;
E5p.error := E5.error;
E5p.estructura := temporal;
E5p.tipo := E5.tipo;
else
E5p.error := true;}
E5p → λ { E5p.error := false; E5p.es_funcion_laboratorio:= false;}
E4 → *E3*{ E4.es_funcion_laboratorio := E3.es_funcion_laboratorio;
E4.error := E3.error;
E4.estructura := E3.estructura;
E4.tipo := E3.tipo;
E4p.estructura := E3.estructura;
E4p.tipo := E3.tipo;
} E4p{
if (E4.es_funcion_laboratorio \/ E4.error)
E4.es_funcion_laboratorio := E4p.es_funcion_laboratorio;
E4.error := E4p.error;
E4.estructura := E4p.estructura;
E4.tipo := E4p.tipo; }
E4p → { E4.es_funcion_laboratorio := false;
E4.estructura := E4p.estructura;
E4.tipo := E4p.tipo;
} *E4*{if (E4.es_funcion_laboratorio = false /\ ((E4.tipo = entero \/ E4.tipo = cadena) /\ (E4p.tipo = entero \/ E4p.tipo = cadena))
E4p.es_funcion_laboratorio := E4.es_funcion_laboratorio;
E4p.error := E4.error;
E4p.estructura := temporal;
if (E4p.tipo = cadena \/ E4.tipo = cadena)
E4p.tipo := cadena;
else
E4p.tipo := entero;
else
E4p.error := true;}
E4p → - { E4.es_funcion_laboratorio := false;
E4.estructura := E4p.estructura;
E4.tipo := E4p.tipo;
}*E4*
{if (E4.es_funcion_laboratorio = false /\ E4.tipo = entero /\ E4p.tipo = entero)
E4p.es_funcion_laboratorio := E4.es_funcion_laboratorio;
E4p.error := E4.error;
E4p.estructura := temporal;
E4p.tipo := E4.tipo;
else
E4p.error := true;}

E4p → λ { E4p.error := false; E4p.es_funcion_laboratorio:= false;}
E3 → *E2*{ E3.es_funcion_laboratorio := E2.es_funcion_laboratorio;
E3.error := E2.error;
E3.estructura := E2.estructura;
E3.tipo := E2.tipo;
E3p.estructura := E2.estructura;
E3p.tipo := E2.tipo;
} E3p{
if (E3.es_funcion_laboratorio ∨ E3.error)
E3.es_funcion_laboratorio := E3p.es_funcion_laboratorio;
E3.error := E3p.error;
E3.estructura := E3p.estructura;
E3.tipo := E3p.tipo; }
E3p → * { E3.es_funcion_laboratorio := false;
E3.estructura := E3p.estructura;
E3.tipo := E3p.tipo;
} *E3*{if (E3.es_funcion_laboratorio = false /\ E3.tipo = entero /\ E3p.tipo = entero)
E3p.es_funcion_laboratorio := E3.es_funcion_laboratorio;
E3p.error := E3.error;
E3p.estructura := temporal;
E3p.tipo := E3.tipo;
else
E3p.error := true;}
E3p → / { E3.es_funcion_laboratorio := false;
E3.estructura := E3p.estructura;
E3.tipo := E3p.tipo;
} *E3*
{if (E3.es_funcion_laboratorio = false /\ E3.tipo = entero /\ E3p.tipo = entero)
E3p.es_funcion_laboratorio := E3.es_funcion_laboratorio;
E3p.error := E3.error;
E3p.estructura := temporal;
E3p.tipo := E3.tipo;
else
E3p.error := true;}
E3p → % { E3.es_funcion_laboratorio := false;
E3.estructura := E3p.estructura;
E3.tipo := E3p.tipo;
} *E3*
{ if (E3.es_funcion_laboratorio = false /\ E3.tipo = entero /\ E3p.tipo = entero)
E3p.es_funcion_laboratorio := E3.es_funcion_laboratorio;
E3p.error := E3.error;
E3p.estructura := temporal;
E3p.tipo := E3.tipo;
else
E3p.error := true;}

E3p → λ { E3p.error := false; E3p.es_funcion_laboratorio:= false;}
E2 → *NOT*{ E1.es_funcion_laboratorio := false;} *E1*
{ if (E1.es_funcion_laboratorio = false /\ E1.tipo =booleano)
E2.es_funcion_laboratorio := E1.es_funcion_laboratorio;
E2.error := E1.error;
E2.estructura := E1.estructura;
E2.tipo := E1.tipo;
else
E2.error := true;}
E2 → ++{ E1.es_funcion_laboratorio := false;} *E1* { if
(E1.es_funcion_laboratorio = false /\ E1.estructura = identificador /\ E1.tipo
=entero)
E2.es_funcion_laboratorio := E1.es_funcion_laboratorio;
E2.error := E1.error;
E2.estructura := E1.estructura;
E2.tipo := E1.tipo;
else
E2.error := true;}
E2 → --{ E1.es_funcion_laboratorio := false;} *E1*{
if (E1.es_funcion_laboratorio = false /\ E1.estructura = identificador /\ E1.tipo
=entero)
E2.es_funcion_laboratorio := E1.es_funcion_laboratorio;
E2.error := E1.error;
E2.estructura := E1.estructura;
E2.tipo := E1.tipo;
else
E2.error := true;}
E2 → +{ E1.es_funcion_laboratorio := false;} *E1*
{ if(E1.tipo = entero)
E2.es_funcion_laboratorio := E1.es_funcion_laboratorio;
E2.error := E1.error;
E2.estructura := temporal;
E2.tipo := E1.tipo;
else
E2.error:= true; }
E2 → -{ E1.es_funcion_laboratorio := false; } *E1*
{ if(E1.tipo = entero)
E2.es_funcion_laboratorio := E1.es_funcion_laboratorio;
E2.error := E1.error;
E2.estructura := temporal;
E2.tipo := E1.tipo;
else
E2.error:= true; }
E2 → *E1*{ E2.es_funcion_laboratorio := E1.es_funcion_laboratorio;
E2.error := E1.error;
E2.estructura := E1.estructura;

E2.tipo := E1.tipo;
E2p.estructura := E1.estructura;
E2p.tipo := E1.tipo;
} E2p{
if (E2.es_funcion_laboratorio ∨ E2.error)
E2.es_funcion_laboratorio := E2p.es_funcion_laboratorio;
E2.error := E2p.error;
E2.estructura := E2p.estructura;
E2.tipo := E2p.tipo; }
E2p → ++ {if (E2p.estructura != identificador ∨ E2p.tipo!=entero)
E2p.error := true;
else
E2p.error := false;
E2p.es_funcion_laboratorio:= false;}
E2p → -- {if (E2p.estructura != identificador ∨ E2p.tipo!=entero)
E2p.error := true;
else
E2p.error := false;
E2p.es_funcion_laboratorio:= false;}
E2p → λ { E2p.error := false; E2p.es_funcion_laboratorio:= false;}
E1 → *id Pf* {
E1.error := Pf.error;
if (Pf.num_parametros > 0)
if (obtenerTS(id).estructura = funcion)
For (i=0; i< Pf.num_parametros; i++)
if Pf[i].tipo ¡= obtenerTS(id).parametros[i].tipo;
E1.error = true;
else if (obtenerTS(id).estructura = funcion_laboratorio)
if (obtenerTS(id).nombre =Mix)
if (Pf.num_parametros.Count <= 1)
E1.error = true;
For (i=0; i< Pf.num_parametros; i++)
If Pf[i].tipo ¡= cadena
E1.error = true;
else if (obtenerTS(id).nombre =Delete)
if (Pf.num_parametros.Count <= 0)
E1.error = true;
For (i=0; i< Pf.num_parametros; i++)
If (Pf[i].tipo ¡= cadena)
E1.error = true;
else
if (Pf.num_parametros =
obtenerTS(id).parametros.num_parametros)
E1.error = true;
For (i=0; i< Pf.num_parametros; i++)
Pf[i].tipo = obtenerTS(id).parametros[i].tipo;

if (obtenerTS(id).nombre =Cut)
E1.error = not ES_ADNSimple(Pf[1]);
E1.error = not ES_ADNSimple(Pf[2]);
else if (obtenerTS(id).nombre =Extract_Subsecuence)
E1.error = not ES_ADNSimple(Pf[1]);
if (obtenerTS(id).tipo = funcion)
E1.error = true;
else if (obtenerTS(id).tipo ¡= entero /\ obtenerTS(id).tipo ¡= cadena /\ obtenerTS(id).tipo ¡= booleano)
E1.error = true;
else if (Pf.num_parametros > 0 \/ obtenerTS(id).estructura = funcion \/ obtenerTS(id).estructura = funcion_laboratorio)
E1.error = true;
}
E1 → *(Ek)* { E1.tipo := Ek.tipo; E1.estructura := Ek.estructura; }
E1 → *numero* { E1.tipo := entero; E1.lugar := numero; E1.estructura := estático; E1.es_funcion_laboratorio:= false;}
E1 → *cadena* { E1.tipo := cadena; E1.lugar := cadena; E1.estructura := estático; E1.es_funcion_laboratorio:= false;}
E1 → *TRUE* { E1.tipo := booleano; E1.lugar := true; E1.estructura := estático; E1.es_funcion_laboratorio:= false;}
E1 → *FALSE* { E1.tipo := booleano; E1.lugar := false; E1.estructura := estático; E1.es_funcion_laboratorio:= false;}
Pf → *(Lpf)* { Pf.error := Lpf.error; Pf.parametros := Lpf.parametros; Pf.num_parametros := Lpf.num_parametros; }
Pf → *λ*{ Pf.error := false; Pf.num_parametros := 0; }
Lpf → { Ek.es_funcion_laboratorio := false;} *Ek* { Lpf.error := Ek.es_funcion_laboratorio; Lpf.error := Ek.error; }*Lpfp* { Lpf.error := Lpfp.error; Lpf.parametros := Lpfp.parametros; Añadir(Ek, Lpf.parametros); Lpf.num_parametros := Lpfp.num_parametros +1;}
Lpf → *λ*{ Lpf.error := false; Lpf.num_parametros := 0; }
Lpfp → , { Ek.es_funcion_laboratorio := false;} *Ek*{ Lpfp.error := Ek.es_funcion_laboratorio; Lpfp.error := Ek.error; } $Lpfp_1$ { Lpfp.error := $Lpfp_1$.error; Lpfp.parametros := $Lpfp_1$.parametros; Añadir(Ek, Lpfp.parametros); Lpfp.num_parametros := $Lpfp_1$.num_parametros +1;}
Lpfp → *λ*{ Lpfp.error := false; Lpfp.num_parametros := 0; }
Ldfsalida → *id* { insertarTS(id).estructura := identificador; insertarTS(id).tipo := ninguno;} *Ldfsalidap*{ Ldfsalida.error := Ldfsalidap.error; Ldfsalida.parametros := Ldfsalidap.parametros; Añadir(obtenerTS(id), Ldfsalida.parametros); Ldfsalida.num_parametros := Ldfsalidap.num_parametros + 1; }
Ldfsalidap → ,*id* { insertarTS(id).estructura := identificador; insertarTS(id).tipo := ninguno;} $Ldfsalidap_1$ { Ldfsalidap.error := $Ldfsalidap_1$.error; Ldfsalidap.parametros := $Ldfsalidap_1$.parametros; Añadir(obtenerTS(id), Ldfsalidap.parametros); Ldfsalidap.num_parametros := $Ldfsalidap_1$.num_parametros + 1;}
Ldfsalidap → *λ*{ Ldfsalidap.error := false; Ldfsalidap.num_parametros := 0;}

Ldf → *INTEGER id*{ insertarTS(id).estructura := identificador; insertarTS(id).tipo := entero; insertarTS(id).lugar:= 0; } *Ldfp* { Ldf.parametros := Ldfp.parametros; Añadir(obtenerTS(id), Ldf.parametros); Ldf.num_parametros := Ldfp.num_parametros + 1; Ldf.error := Ldfp.error;}
Ldf → *BOOLEAN id*{ insertarTS(id).estructura := identificador; insertarTS(id).tipo := booleano; insertarTS(id).lugar:= false;} *Ldfp* { Ldf.parametros := Ldfp.parametros; Añadir(obtenerTS(id), Ldf.parametros); Ldf.num_parametros := Ldfp.num_parametros + 1; Ldf.error := Ldfp.error;}
Ldf → *STRING id*{ insertarTS(id).estructura := identificador; insertarTS(id).tipo := cadena; insertarTS(id).lugar := "";}*Ldfp* {Ldf.parametros := Ldfp.parametros; Añadir(obtenerTS(id), Ldf.parametros); Ldf.num_parametros := Ldfp.num_parametros + 1; Ldf.error := Ldfp.error; }
Ldf → *λ* {Ldf.error := false; Ldf.num_parametros := 0;}
Ldfp →, *INTEGER id*{ insertarTS(id).estructura := identificador; insertarTS(id).tipo := entero; insertarTS(id).lugar := 0;} *Ldfp*$_1$ { Ldfp.parametros := Ldfp$_1$.parametros; Añadir(obtenerTS(id), Ldfp.parametros); Ldfp.num_parametros := Ldfp$_1$.num_parametros + 1; Lpfp.error := Ldfp$_1$.error;}
Ldfp →, *BOOLEAN id*{ insertarTS(id).estructura := identificador; insertarTS(id).tipo := booleano; insertarTS(id).lugar := false;} *Ldfp*$_1$ {Ldfp.parametros := Ldfp$_1$.parametros; Añadir(obtenerTS(id), Ldfp.parametros); Ldfp.num_parametros := Ldfp$_1$.num_parametros + 1; Lpfp.error := Ldfp$_1$.error;}
Ldfp →, *STRING id*{ insertarTS(id).estructura := identificador; insertarTS(id).tipo := cadena; insertarTS(id).lugar := "";}*Ldfp*$_1$ {Ldfp.parametros := Ldfp$_1$.parametros; Añadir(obtenerTS(id), Ldfp.parametros); Ldfp.num_parametros := Ldfp$_1$.num_parametros + 1; Lpfp.error := Ldfp$_1$.error;}
Ldfp → *λ* { Ldfp.error := false; Ldfp.num_parametros := 0;}
Lpfsalida → *id Lpfsalidap*

```
{ Lpfsalida.error := Lpfsalidap.error;
if (obtenerTS(id).estructura = identificador)
        Lpfsalida.parametros := Lpfsalidap.parametros;
        Añadir(obtenerTS(id), Lpfsalida.parametros);
        Lpfsalida.num_parametros := Lpfsalidap.num_parametros
                + 1;
    else
        Lpfsalida.error := true;}
```

Lpfsalidap →, *Lpfsalida* { Lpfsalidap.error := Lpfsalida.error; Lpfsalidap.num_parametros := Lpfsalida.num_parametros, Lpfsalidap.parametros := Lpfsalida.parametros; }
Lpfsalidap → *λ* { Lpfsalidap.error := false; := Lpfsalidap.num_parametros := 0;}

4.9.4 Código intermedio

Generamos cuartetos, debido a que son más intuitivos para al ver el código intermedio sepamos lo que tenemos que hacer con él. También nos sirvió para tener un mayor control y realizar comprobaciones semánticas.

P → *FUNCTION Pd P*$_1$

P.codigo := P_1.codigo

P → id Pp P1

P.codigo := Pp.codigo || P_1.codigo

P → (Lpfsalida) = id (Lpf) P1

P.codigo := P_1.codigo

P → λ

P.codigo := ‘’

Pp → = Ek

if (Ek.estructura != function_laboratorio)

Pp.codigo := Ek.codigo|| gen(Pp.lugar ‘:=’ Ek.lugar);

Pp → += Ek

Pp.codigo := Ek.codigo|| gen(Pp.lugar ‘:=’ Pp.lugar ‘+’ Ek.lugar);

Pp → -= Ek

Pp.codigo := Ek.codigo|| gen(Pp.lugar ‘:=’ Pp.lugar ‘-’ Ek.lugar);

*Pp → *= Ek*

Pp.codigo := Ek.codigo|| gen(Pp.lugar ‘:=’ Pp.lugar ‘*’ Ek.lugar);

Pp → %= Ek

Pp.codigo := Ek.codigo|| gen(Pp.lugar ‘:=’ Pp.lugar ‘%’ Ek.lugar);

Pp → /= Ek

Pp.codigo := Ek.codigo|| gen(Pp.lugar ‘:=’ Pp.lugar ‘/’ Ek.lugar);

Pp → (Lpf) P

if (Pp.estructura = function)

Pp.codigo := gen(‘call’ Pp.nombre) || Lpf.codigo || P.codigo

else if (Pp.estructura = funcion_laboratorio)

Pp.codigo := gen(‘call’ Pp.nombre) || Lpf.codigo || P.codigo

Pd → (Ldfsalida) id (Ldf) C END

Pd → id Pd1

Pd1 → (Ldf) C END

Pd1 → id (Ldf) C END

C → IF Eb THEN C1 Oelse END C2

Eb.falsa := nuevaEtiqueta();

C.codigo := Eb.codigo || gen (‘if’ Eb.lugar ‘=‘ ‘false’ ‘goto’ Eb.falsa) || C_1.codigo || gen(‘goto’ C_2.fin) || gen(Eb.falsa ‘:’) || Oelse.codigo || C_2.codigo

C → FOR id = Ea TO Ea1 DO C1 END C2

Ea.inicio := nuevaEtiqueta();

C_2.fin := nuevaEtiqueta();

temp := nuevoTemp();

C.codigo := gen(obtenerTS(id).lugar ‘:=’ Ea.lugar) || gen(Ea.inicio ‘:’) || gen(temp ‘:=’ obtenerTS(id).lugar ‘=’ Ea_1.lugar) || gen (‘if’ temp ‘=‘ ‘false’ ‘goto’ C_2.fin) || C_1.codigo || gen(obtenerTS(id).lugar ‘:=’ obtenerTS(id).lugar ‘+’ ‘1’) || gen(‘goto’ Ea.inicio) || gen(C_2.fin ‘:’) || C_2.codigo

C → WHILE Eb DO C1 END C2

C.comienzo := nuevaEtiqueta();

C_2.fin := nuevaEtiqueta();

C.codigo := gen(C.comienzo ':') || Eb.codigo || gen ('if' Eb.lugar '=' 'false' 'goto' C_2.fin) || C_1.codigo || gen('goto' C.comienzo) || gen(C_2.fin ':') || C_2.codigo

C → (Lpfsalida) = id (Lpf) C_1

C.codigo := Lpfsalida.codigo || gen('call' obtenerTS(id).nombre) || Lpf.codigo || C_1.codigo

C → id Cp C_1

C.lugar := obtenerTS(id).lugar
C.codigo := Cp.codigo || C_1.codigo

C → λ

C.codigo := ''

Oelse → ELSE C

Oelse.codigo := C.codigo

Oelse → λ

Oelse.codigo := ''

Cp → = Ek

if (Ek.estructura != function_laboratorio)
Cp.codigo := Ek.codigo|| gen(Cp.lugar ':=' Ek.lugar);

Cp → += Ek

Cp.codigo := Ek.codigo|| gen(Cp.lugar ':=' Cp.lugar '+' Ek.lugar);

Cp → -= Ek

Cp.codigo := Ek.codigo|| gen(Cp.lugar ':=' Cp.lugar '-' Ek.lugar);

*Cp → *= Ek*

Cp.codigo := Ek.codigo|| gen(Cp.lugar ':=' Cp.lugar '*' Ek.lugar);

Cp → %= Ek

Cp.codigo := Ek.codigo|| gen(Cp.lugar ':=' Cp.lugar '%' Ek.lugar);

Cp → /= Ek

Cp.codigo := Ek.codigo|| gen(Cp.lugar ':=' Cp.lugar '/' Ek.lugar);

Cp → ++

Cp.codigo := Cp.codigo || gen (Cp.lugar ':=' Cp.lugar '+' '1')

Cp → --

Cp.codigo := Cp.codigo || gen (Cp.lugar ':=' Cp.lugar '-' '1')

Cp → (Lpf) C

if (Cp.estructura = function)
Cp.codigo := gen('call' Cp.nombre) || Lpf.codigo || C.codigo
else if (Cp.estructura = funcion_laboratorio)
Cp.codigo := gen('call' Cp.nombre) || Lpf.codigo || C.codigo

Ea → Ek

Ea.lugar := Ek.lugar
Ea.codigo := Ek.codigo

Eb → Ek

Eb.lugar := Ek.lugar
Eb.codigo := Ek.codigo

Ek → E8

Ek.lugar := E8.lugar
Ek.codigo := E8.codigo

E8 → *E7 E8p*
E8.lugar := E7.lugar || E8p.lugar
E8.codigo := E7.codigo || E8p.codigo
E8p → *OR E8*
E8p.lugar := nuevoTemp();
E8p.codigo := E8p.codigo || E8.codigo || gen(E8p.lugar ':=' E8p.lugar 'OR' E8.lugar)
E8p → *λ*
E8p.codigo := ''
E7 → *E6 E7p*
E7.lugar := E6.lugar || E7p.lugar
E7.codigo := E6.codigo || E7p.codigo
E7p → *AND E7*
E7p.lugar := nuevoTemp();
E7p.codigo := E7p.codigo || E7.codigo || gen(E7p.lugar ':=' E7p.lugar 'AND' E7.lugar)
E7p → *λ*
E7p.codigo := ''
E6 → *E5 E6p*
E6.lugar := E5.lugar || E6p.lugar
E6.codigo := E5.codigo || E6p.codigo
E6p → *EQUAL E6*
E6p.lugar := nuevoTemp();
E6p.codigo := E6p.codigo || E6.codigo || gen(E6p.lugar ':=' E6p.lugar '=' E6.lugar)
E6p → *UNEQUAL E6*
E6p.lugar := nuevoTemp();
E6p.codigo := E6p.codigo || E6.codigo || gen(E6p.lugar ':=' E6p.lugar '!=' E6.lugar)
E6p → *λ*
E6p.codigo := ''
E5 → *E4 E5p*
E5.lugar := E4.lugar || E5p.lugar
E5.codigo := E4.codigo || E5p.codigo
E5p → *> E5*
E5p.lugar := nuevoTemp();
E5p.codigo := E5p.codigo || E5.codigo || gen(E5p.lugar ':=' E5p.lugar '>' E5.lugar)
E5p → *< E5*
E5p.lugar := nuevoTemp();
E5p.codigo := E5p.codigo || E5.codigo || gen(E5p.lugar ':=' E5p.lugar '<' E5.lugar)
E5p → *<= E5*
E5p.lugar := nuevoTemp();
E5p.codigo := E5p.codigo || E5.codigo || gen(E5p.lugar ':=' E5p.lugar '<=' E5.lugar)

E5p → >= *E5*
E5p.lugar := nuevoTemp();
E5p.codigo := E5p.codigo || E5.codigo || gen(E5p.lugar ':=' E5p.lugar '>=' E5.lugar)

E5p → λ
E5p.codigo := ''

E4 → *E3 E4p*
E4.lugar := E3.lugar || E4p.lugar
E4.codigo := E3.codigo || E4p.codigo

E4p → + *E4*
E4p.lugar := nuevoTemp();
E4p.codigo := E4p.codigo || E4.codigo || gen(E4p.lugar ':=' E4p.lugar '+' E4.lugar)

E4p → - *E4*
E4p.lugar := nuevoTemp();
E4p.codigo := E4p.codigo || E4.codigo || gen(E4p.lugar ':=' E4p.lugar '-' E4.lugar)

E4p → λ
E4p.codigo := ''

E3 → *E2 E3p*
E3.lugar := E2.lugar || E3p.lugar
E3.codigo := E2.codigo || E3p.codigo

E3p → * *E3*
E3p.lugar := nuevoTemp();
E3p.codigo := E3p.codigo || E3.codigo || gen(E3p.lugar ':=' E3p.lugar '*' E3.lugar)

E3p → / *E3*
E3p.lugar := nuevoTemp();
E3p.codigo := E3p.codigo || E3.codigo || gen(E3p.lugar ':=' E3p.lugar '/' E3.lugar)

E3p → % *E3*
E3p.lugar := nuevoTemp();
E3p.codigo := E3p.codigo || E3.codigo || gen(E3p.lugar ':=' E3p.lugar '%' E3.lugar)

E3p → λ
E3p.codigo := ''

E2 → *NOT E1*
E2.lugar := nuevoTemp();
E2.codigo := E1.codigo || gen(E2.lugar ':=' 'not' E1.lugar)

E2 → ++ *E1*
E2.lugar := E1.lugar;
E2.codigo := gen (E1.lugar ':=' E1.lugar '+' '1') || E1.codigo

E2 → -- *E1*
E2.lugar := E1.lugar;
E2.codigo := gen (E1.lugar ':=' E1.lugar '-' '1') || E1.codigo

E2 → *+ E1*
E2.lugar := nuevoTemp();
E2.codigo := E1.codigo || gen (E2.lugar ':=' '0' '+' E1.lugar)
E2 → *- E1*
E2.lugar := nuevoTemp();
E2.codigo := E1.codigo || gen (E2.lugar ':=' '0' '-' E1.lugar)
E2 → *E1 E2p*
E2.lugar := E1.lugar || E2p.lugar
E2.codigo := E1.codigo || E2p.codigo
E2p → ++
E2p.codigo := E2p.codigo || gen (E2p.lugar ':=' E2p.lugar '+' '1')
E2p → --
E2p.codigo := E2p.codigo || gen (E2p.lugar ':=' E2p.lugar '-' '1')
E2p → λ
E2p.codigo := ''
E1 → *id Pf*
if (obtenerTS(id).estructura = funcion)
{
E1.codigo := gen ('call' obtenerTS(id).nombre) || Pf.codigo
if (Pf.num_parametros = 1 ∨ Pf.num_parametros = 0)
{
E1.lugar := nuevoTemp();
E1.codigo := E1.codigo || gen(E1.lugar ':='
obtenerTS(id).lugar)
}
}
else if (id.estructura = funcion_laboratorio)
E1.codigo := gen ('call' obtenerTS(id).nombre) || Pf.codigo
E1 → *(Ek)*
E1.lugar := Ek.lugar
E1.codigo := Ek.codigo
E1 → *numero*
E1.lugar := nuevoTemp();
E1.codigo := gen (E1.lugar ':=' numero);
E1 → *cadena*
E1.lugar := nuevoTemp();
E1.codigo := gen (E1.lugar ':=' cadena);
E1 → *TRUE*
E1.lugar := nuevoTemp();
E1.codigo := gen (E1.lugar ':=' 'true');
E1 → *FALSE*
E1.lugar := nuevoTemp();
E1.codigo := gen (E1.lugar ':=' 'false');
Pf → *(Lpf)*
Pf.codigo := Lpf.codigo
Pf → λ

Pf.codigo := ‘’
Lpf → *Ek Lpfp*
Lpf.codigo := Ek.codigo || Lpfp.codigo || gen (‘param’ Ek.lugar);
Lpf → λ
Lpf.codigo := ‘’
Lpfp → , *Ek* $Lpfp_1$
Lpfp.codigo := Ek.codigo || $Lpfp_1$.codigo || gen (‘param’ Ek.lugar);
Lpfp → λ
Lpfp.codigo := ‘’
Ldfsalida → *id Ldfsalidap*
Ldfsalidap → ,*id Ldfsalidap*
Ldfsalidap → λ
Ldf → *INTEGER id Ldfp*
Ldf → *BOOLEAN id Ldfp*
Ldf → *STRING id Ldfp*
Ldf → λ
Ldfp → , *INTEGER id* $Ldfp_1$
Ldfp → , *BOOLEAN id* $Ldfp_1$
Ldfp → , *STRING id* $Ldfp_1$
Ldfp → λ
Lpfsalida → *id Lpfsalidap*
Lpfsalida.codigo := gen (‘pop’ obtenerTS(id).lugar) || Lpfsalidap.codigo
Lpfsalidap → , *Lpfsalida*
Lpfsalidap.codigo := Lpfsalida.codigo
Lpfsalidap → λ
Lpfsalidap.codigo := ‘’

4.9.5 Tabla de símbolos

Como la ejecución tiene que ser con el lenguaje c#, necesitamos una estructura donde estuviese la compilación, pero al mismo tiempo las variables de memoria de tiempo de ejecución, por eso la tabla de símbolos es un poco diferente a las tablas descritas en la teoría. En ella estarán el valor de las variables, tanto locales, como temporales.

Para utilizar la tabla de símbolos como memoria en tiempo de ejecución tuvimos que crear una tabla de símbolos de ejecución por cada llamada de función. Bastante similar a la zona de memoria en tiempo de ejecución.

A continuación detallamos los atributos de la tabla de símbolos.

- Token. Corresponde al token que le ha pasado el analizador léxico, contiene entre ellos el nombre de la variable, la tabla de símbolos a la que pertenece y su posición en ella.

- Valor. Representa el valor del elemento.

- tipo. Corresponde al tipo de la variable, puede ser: booleano, entero, cadena, real, funcion, ninguno. El más raro es el de función y es que un tipo, al ser función indica que tiene más de un parámetro.
- Estructura. Es la estructura que tiene el elemento. Determina que es. Sus posibles valores son: identificador, funcion, funcion_laboratorio, temporal. Función de laboratorio es para identificar a las funciones de laboratorio que tienen una ejecución particular en la pantalla principal de la aplicación, el temporal es debido que puede ser una variable temporal.
- Parámetros. Es una estructura que contiene el tipo de los parámetros y su nombre.
- tabla_apunta. Se refiere a la tabla a la que apunta la función si es necesario que tenga tabla. (Las funciones de laboratorio no tienen tabla de símbolos).

4.9.6 Ejecución

Esta es la parte que más diferencia tiene con un compilador, debido a que es el propio programa el que tiene que ir ejecutando las instrucciones.

De la zona de memoria sólo tendremos dos partes:

- Instrucciones del código intermedio.
- Tabla de símbolos.

Por cada función tenemos unas instrucciones de código intermedio y una tabla de símbolos que será la que soporte todas las variables y las variables temporales de tiempo de ejecución.

Como es lógico tendremos registros de activación. Cuando se haga una llamada a una función, el registro de activación guardará la posición en que nos hemos quedado ejecutando, y se creará un nuevo registro de activación. Este creará una copia de la tabla de símbolos asociada a ella, para su ejecución. Así garantizamos los ámbitos de las variables.

Desde el compilador solo se podrán ejecutar funciones que no tengan parámetros. Antes de ejecutar dicha función se ejecutarán todas las instrucciones del ámbito global, por lo que, la llamada a la función es como si estuviera al final del código.

Tenemos también dos funciones implementadas. Estas están contenidas en la tabla de símbolos y al ejecutar realizamos las operaciones que deben realizar (son como una llamada a una función).

4.9.7 Función de laboratorio Annealing

La traducción de la función de laboratorio Annealing a algoritmo no ha sido sencilla y también ha sido muy tediosa debido principalmente a que es irresoluble con las técnicas hoy conocidas, solo ha podido limitarse.

El annealing al fin y al cabo te devuelve todas las posibles combinaciones de todas las hebras donde se puedan enganchar. A simple vista parece fácil, pero como puede haber huecos en las hebras lo hace muy complicado porque se pueden enganchar las hebras al final de las cadenas y volverse a enganchar con un pequeño trozo al final.

Un ejemplo de esto es el annealing de la cadena simple AAATTT. Está produciría muchas cadenas, 3 de estas son:

```
AAATTT      AAATTT          AAATTT
TTTAAA        AAATTT          AAATTT
```

Habría muchas más ya que la cadena se podría volver a enganchar con la cadena de ADN, y así seguir de forma infinita.

Como se aprecia es una explosión combinatoria bastante grande, por lo que se opto por limitar el límite mínimo de unión en cadenas al hacer el annealing y el tamaño máximo de cadenas al tratarlo.

Se ha ido acotando las soluciones, primero tratando las cadenas que tuvieses en el tubo de ensayo, y luego las que iban surgiendo de él, hasta ver que no se generaba ninguna más que ya no existiese o que se llegase al límite de tamaño.

En cada iteración se hacían todas las posibilidades de las cadenas simples, de las cadenas de ADN, de las cadenas de ADN simple con las cadenas de ADN normales. Así el trabajo ha sido más estructurado y al encontrar fallos al generar las posibilidades era fácilmente detectable.

4.10 Caso práctico.

Con estos datos ya podemos elaborar protocolos, para ellos nos vamos a referenciar en el experimento de Adleman (Pág 18)

4.10.1 Adleman

Con la ayuda del experimento de Adleman (Pág. 22) realizamos el protocolo para la obtención del camino hamiltoniano.

1- Creación de los vértices. Codificación de las aristas con la información de las aristas. Lo generamos en la Probeta0.

2- Generar todos los caminos en Probeta1.

3- Limitar las soluciones a los que miden 140 (un camino completo). Habrá 4 probetas de resultados, la válida al final es Probeta33.

4- Limitar las soluciones que no pasen por todos los nodos. La solución se contendrá en la Probeta 40.

La función H() y la función AleatoryChaynADN() son unos funciones que han sido definidas en el compilador.

La función H() devuelve en forma de cadena el complementario de la cadena simple de ADN que se le pasa (también esta en forma de cadena).

La función AleatoryChaynADN() devuelve en forma de cadena una cadena aleatoria de ADN, necesita como parámetro de entrada la longitud de la cadena de salida.

```
FUNCTION Adleman()

//****************** Preparando los vértices y aristas ********************
 tamanyo = 10

//preparando los vertices.
 v0p = AleatoryChaynADN(tamanyo )
 v0pp = AleatoryChaynADN(tamanyo )
 v0 = v0p + v0pp

 v1p = AleatoryChaynADN(tamanyo )
 v1pp = AleatoryChaynADN(tamanyo )
 v1 = v1p + v1pp

 v2p = AleatoryChaynADN(tamanyo )
```

```
v2pp = AleatoryChaynADN(tamanyo )
v2 = v2p + v2pp

v3p = AleatoryChaynADN(tamanyo )
v3pp = AleatoryChaynADN(tamanyo )
v3 = v3p + v3pp

v4p = AleatoryChaynADN(tamanyo )
v4pp = AleatoryChaynADN(tamanyo )
v4 = v4p + v4pp

v5p = AleatoryChaynADN(tamanyo )
v5pp = AleatoryChaynADN(tamanyo )
v5 = v5p + v5pp

v6p = AleatoryChaynADN(tamanyo )
v6pp = AleatoryChaynADN(tamanyo )
v6 = v6p + v6pp

// creacion de las aristas
a01 = H(v0pp) + H(v1p)
a03 = H(v0pp) + H(v3p)
a06 = H(v0pp) + H(v6p)
a12 = H(v1pp) + H(v2p)
a13 = H(v1pp) + H(v3p)
a21 = H(v2pp) + H(v1p)
a23 = H(v2pp) + H(v3p)
a32 = H(v3pp) + H(v2p)
a34 = H(v3pp) + H(v4p)
a41 = H(v4pp) + H(v1p)
a45 = H(v4pp) + H(v5p)
a51 = H(v5pp) + H(v1p)
a52 = H(v5pp) + H(v2p)
a56 = H(v5pp) + H(v6p)

AddChaynADN(Probeta0, v0)
AddChaynADN(Probeta0, v1)
AddChaynADN(Probeta0, v2)
AddChaynADN(Probeta0, v3)
AddChaynADN(Probeta0, v4)
AddChaynADN(Probeta0, v5)
AddChaynADN(Probeta0, v6)

AddChaynADN(Probeta0, a01)
AddChaynADN(Probeta0, a03 )
```

```
AddChaynADN(Probeta0, a06 )
AddChaynADN(Probeta0, a12 )
AddChaynADN(Probeta0, a13 )
AddChaynADN(Probeta0, a21 )
AddChaynADN(Probeta0, a23 )
AddChaynADN(Probeta0, a32 )
AddChaynADN(Probeta0, a34 )
AddChaynADN(Probeta0, a41 )
AddChaynADN(Probeta0, a45 )
AddChaynADN(Probeta0, a51 )
AddChaynADN(Probeta0, a52 )
AddChaynADN(Probeta0, a56 )

//***************** Generar todos los caminos. ******************
Probeta0 = "Probeta0"
Probeta1 = "Probeta1"
Probeta1 = Anneling(Probeta0)

//***************** Limitar las soluciones a los que miden 140 ***********
Probeta30 = "Probeta30"
Probeta31 = "Probeta31"
Probeta32 = "Probeta32"
Probeta33 = "Probeta33"

(Probeta30 , Probeta31 ) = Length_Separating(Probeta1, 140)
(Probeta32 , Probeta33 ) = Length_Separating(Probeta30 , 141)

//***************** Limitar las soluciones que no pasen por todos los nodos ***
Probeta40 = "Probeta40"
Probeta41 = "Probeta41"
Probeta42 = "Probeta42"
(Probeta40 , Probeta41 ) = Extract_Subsecuence(Probeta33 ,v0 )
Delete(Probeta41 )
(Probeta41 , Probeta42 ) = Extract_Subsecuence(Probeta40 ,v1 )
Delete(Probeta42 )
(Probeta40 , Probeta42 ) = Extract_Subsecuence(Probeta41 ,v2 )
Delete(Probeta42 )
(Probeta41 , Probeta42 ) = Extract_Subsecuence(Probeta40 ,v3 )
Delete(Probeta42 )
(Probeta40 , Probeta42 ) = Extract_Subsecuence(Probeta41 ,v4 )
Delete(Probeta42 )
(Probeta41 , Probeta42 ) = Extract_Subsecuence(Probeta40 ,v5 )
Delete(Probeta42 )
(Probeta40 , Probeta42 ) = Extract_Subsecuence(Probeta41 ,v6 )
Delete(Probeta42 )
```

```
END
```

Como se puede apreciar el protocolo es muy sencillo de seguir, las variables v son los vértices el número de al lado corresponde al número del vértice, el p y pp, representan la primera parte de la cadena de ADN y la segunda respectivamente.

Las aristas son representadas con la letra a, y los números representan el primero de donde sale, y el segundo a donde llega.

La función Anneling realiza el annealing de la probeta con los vértices y aristas codificados. Así obtenemos todas las posibles combinaciones de las uniones.

La función Length_Separating separa las cadenas que superan esa cantidad de nucleótidos, así filtramos la solución con longitud 140.

La función Extract_Subsecuence selecciona las soluciones que contienen los vértices.

La función Delete borra las probetas que no necesitamos y que se han obtenido del proceso de las funciones definidas en el laboratorio.

Ejecutando el protocolo obtenemos los siguientes resultados:

V0 = CGATATCTCGTTCAATGGAC
V1 = TCTTCCGGTTCGGCCCATAA
V2 = ACCCTGGTCTCTGGAGGGTA
V3 = TGACAGGAGGCCCTGGGAGA
V4 = GTAACGAGCCTGGTCCAGTG
V5 = GACTCCTCTATACCTCTAAG
V6 = AAGAGCAGGTCGATCCAACA
a01 = AAGTTACCTGAGAAGGCCAA
a03 = AAGTTACCTGACTGTCCTCC
a06 = AAGTTACCTGTTCTCGTCCA
a12 = GCCGGGTATTTGGGACCAGA
a13 = GCCGGGTATTACTGTCCTCC
a21 = GACCTCCCATAGAAGGCCAA
a23 = GACCTCCCATACTGTCCTCC
a32 = GGGACCCTCTTGGGACCAGA
a34 = GGGACCCTCTCATTGCTCGG
a41 = ACCAGGTCACAGAAGGCCAA
a45 = ACCAGGTCACCTGAGGAGAT
a51 = ATGGAGATTCAGAAGGCCAA
a52 = ATGGAGATTCTGGGACCAGA

a56 = ATGGAGATTCTTCTCGTCCA

Cadena final:

```
CGATATCTCGTTCAATGGACTCTTCCGGTTCGGCCCATAAACCCTGGTCT
              AAGTTACCTGAGAAGGCCAAGCCGGGTATTTGGGACCAGA
*CTGGAGGGTATGACAGGAGGCCCTGGGAGAGTAACGAGCCTGGTCCAGTG
 GACCTCCCATACTGTCCTCCGGGACCCTCTCATTGCTCGGACCAGGTCAC
*GACTCCTCTATACCTCTAAGAAGAGCAGGTCGATCCAACA
 CTGAGGAGATATGGAGATTCTTCTCGTCCA
```

Se ha tenido que poner en varías líneas la cadena ya que era muy larga y no cabía. El asterisco indica que es continuación de la cadena anterior.

Como se puede comprobar la cadena superior es la cadena formada por v0, v1, v2, v3, v4, v5 y v6. Y sus complementarios son las aristas a01, a12, a23, a34, a45, a56. Interpretando está solución obtenemos la solución al camino hamiltoniano para el grafo indicado (página 20).

5 Conclusiones

Las conclusiones obtenidas han sido un tanto inesperadas. La primera es que el desarrollo de una aplicación sin unos claros conocimientos de lo que se quiere realizar va a conllevar una inversión de tiempo demasiado considerable. Sobre todo porque es difícil ponerse de acuerdo con lo que uno quiere y con lo que puede realizar.

Algunos objetivos todavía son claramente invaluables, porque la sencillez e interpretación del protocolo ha sido realizada con el máximo interés, pero puede que para alguien que no sepa programar, le resulte poco intuitivo.

Por lo menos al desarrollar el proyecto, he podido dar a conocer las ideas que aquí se han manejado y lo interesante que sería investigar sobre estas posibilidades que el ADN nos ofrece. Deseo que esta curiosidad también lo tenga los lectores de este libro.

6 Bibliografía

Adleman 1994 **L.Adleman:**
Molecular computation of solutions to combinatorial problems Science, Nov. 1994.

Microsoft 2001 *Microsoft Developer Network (MSDN) Library.*
Microsoft. Octubre 2001.

Pérez 2003 **Mario de J. Pérez Jiménez, Fernando Sancho Caparrini:**
Maquinas moleculares basadas en ADN.
Colección de divulgación cientifica. Sevilla, 2003.

Ryu 1986 **Will Ryu:**
DNA Computing: A primer

Pisanti 1998 **N.Pisanti:**
A survey on DNA computing.
EATCS Bulletin nº64. 1998.

Paun 1998 **G.Paun, G. Rozenberg, A. Salomaa:**
DNA Computing Paradigms.
Editorial Springer-Verlay. 1998.

Crespo 1982 **Crespo López, Decoroso**
Informática teórica Segunda parte

Crespo 1982 **Crespo López, Decoroso**
Informática teórica Primera parte

Symposium 2004 **Symposium on Graph Drawing (11º. 2003. Perugia, Italia)**
Graph drawing : 11th international symposium, GD 2003, Perugia, Italy, September 21-24, 2003 : revised papers
Editorial Springer-Verlay. 2004.

Hernández 2003 **Hernández Peñalver, Gregorio**
Grafos : teoría y algoritmos
Fundación General de la U.P.M. 2003.

Kakde 2003 **Kakde, O. G.**
Algorithms for compiler design
Editorial Charles River Media. 2003.

Allen 2002 **Allen, Randy**
Optimizing compilers for modern architectures : a dependence-based approach
Editorial Morgan Kaufmann. 2002.

Louden 1997 **Louden, Kenneth C**
Compiler construction : principles and practice
Editorial PWS Publishing Company. 1997.

Aho 1990 **Aho, Alfred V.**
Compiladores : principios, técnicas y herramientas.
Editorial Addison-Wesley Iberoamericana. 1990.