Trabajo final Parser

Andrés Guido Grillo
Legajo : G-5811/4

Documentacion de ejercicio

Objetivo

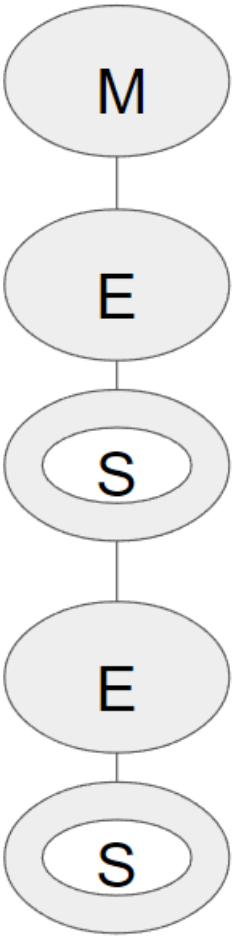
El programa que se tendrá que realizar, será un analizador sintáctico rudimentario: éste leerá un archivo de diccionario (provisto por la Cátedra), tomar´a un archivo de texto de entrada y deberá entregar frases espaciadas de manera que se maximice localmente el tamaño de las palabras indicando en cada caso si fue necesario recuperarse de errores.
La versión rudimentaria del analizador sintáctico que se pide no incluye ningún tipo de predicciones a futuro. Es decir, no debe considerar tomar un palabra de tamaño no maximal para en un futuro encontrar una palabra más larga (que se encuentre en el diccionario).

Input

El archivo de entrada tiene k lineas de texto que representan las frases a espaciar. Considere que los caracteres de las frases podrían ser letras mayúsculas, minúsculas, o una mezcla de ambas.

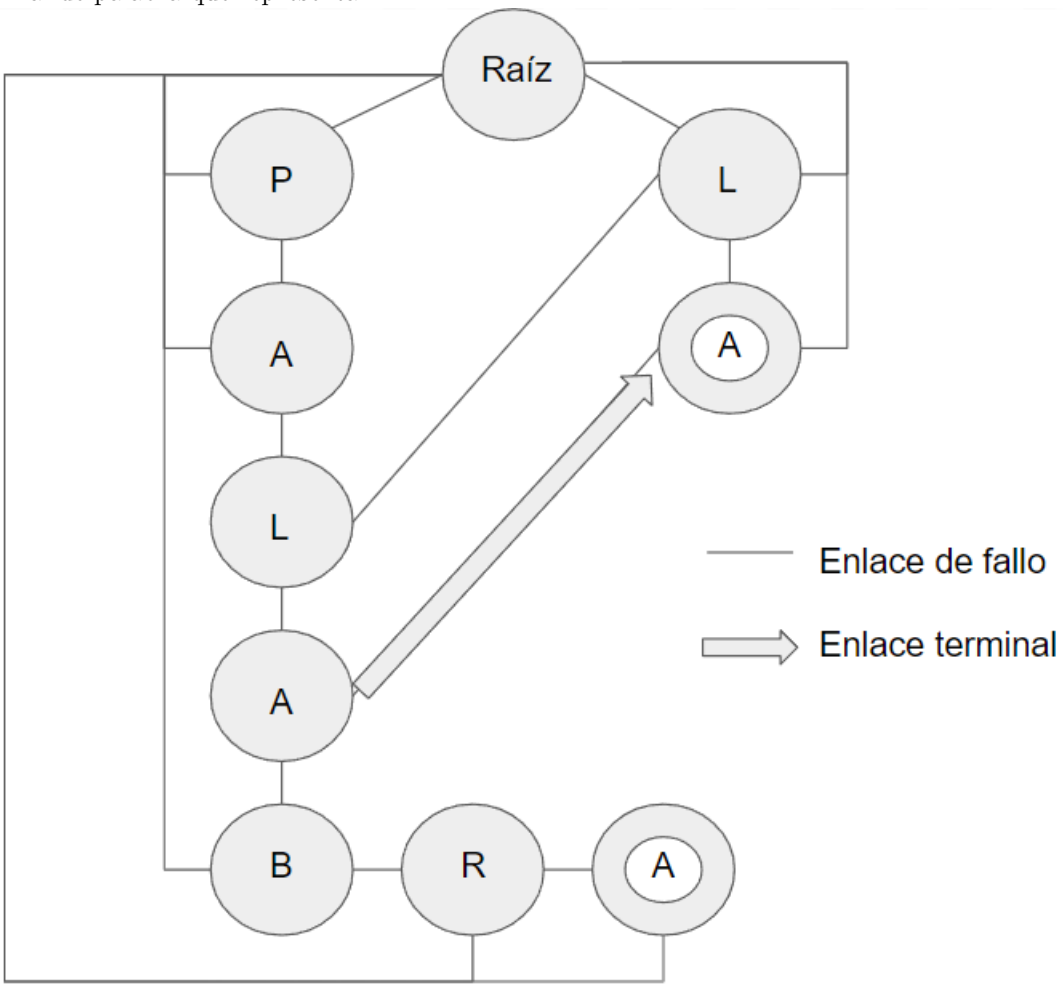
Estructura de datos

El programa hace uso de un arbol general para represantar el diccionario. Cada nodo representa una letra y la posicion de dicha letra en la palabra esta dada por la profundidad del nodo. Un campo adicional no permite saber si la letra es o no es el final de una palabra. Por ejemplo: si el diccionario contuviera las palabras "mes" y "meses" habría un unico camino de nodos que forma mes y tiene marcado el nodo que representa a la "s" como fin de palabra y a su vez ese mismo nodo tiene como hijos el resto de la palabra "meses".



Por lo que pude encontrar al respecto, esta estructura tiene el nombre de trie y me referiré a la estructura con ese nombre apartir de ahora.

El principal problema a la hora de utilizar los trie como forma de representar un diccionario, es que no tienen en cuenta si una palabra contiene una subcadena que es prefijo de otra. Por lo que si quisieramos espaciar correctamente una oración tendríamos que recorrela multiples veces.
Por eso luego de realizar busquedas tratando de optimizar la lectura del trie, encontré el algoritmo Aho-Corasick. Este algoritmo tiene como entrada un trie y retorna un automata de estado finito determinista en donde cada nodo apunta al prefijo más grande al que puede pertenecer. Además cada nodo que contenga un posible final de palabra tendra un enlace terminal que apunta al nodo del final de palabra que representa.



Ejemplo de un trie al que se le aplicó el algoritmo Aho-corasick.

El algoritmo tiene dos invariantes:

- La raíz tiene a si misma como enlace de fallo
- Los hijos inmediatos de la raíz tienen como enlace de fallo a la raiz.

Y para el resto de nodos, la regla es:
Dado un nodo N que representa la letra X con su padre enlazado al nodo P:

- Si P tiene un hijo que representa a X entonces se enlaza a N a ese hijo
- En caso contrario se reemplaza P por su enlace de fallo
- Se repite el proceso hasta llegar a la raiz o enlazar a N.
- Si se llega a la raíz entonces N tiene se enlaza a la raíz.

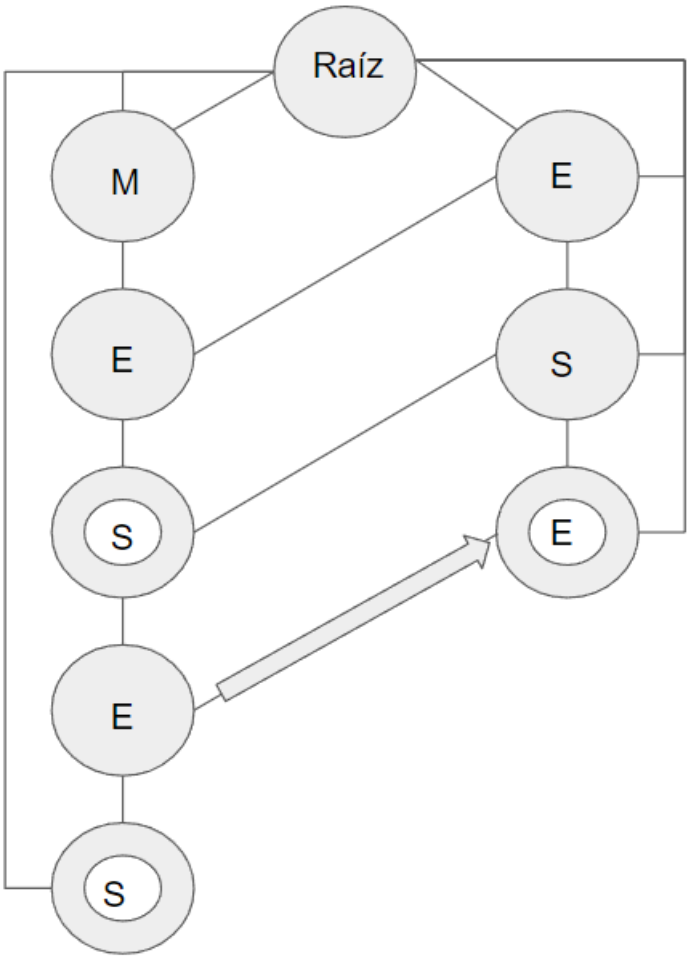
Por lo tanto la estrucutra final es la siguiente:

```
1  typedef struct AEFND {
2   int profundidad;
3   tipoEstado letraFinal;
4   //no hacemos diferencia entre minusculas y mayusculas
5   struct AEFND *siguientes[26];
6   struct AEFND *enlaceFallo;
7   struct AEFND *enlaceTerminal;
8  } _Diccionario;
9
10 typedef _Diccionario *Diccionario;
```

El problema con este algoritmo es que no tiene encuenta si esta cortando un palabra que ya aparecio para poder crear los enlaces. Por ejemplo, supongamos que tenemos ingresado en nuestro diccionario las palabras:

- Mes
- Meses
- Ese

Entonces el resultado de aplicar el algotirmo sería el siguiente:



El problema con este arbol es que enlaza la segunda "e" de "meses" con "ese" pero hacer eso implica que está utilizando la subcadena "ese" en "meses" para formarla pero "mes" aparece antes por lo tanto si llegasemos en una lectura hasta ese nodo, nos redirigiría erroneamente.
Para poder evitar este problema se debe agregar invariante al algoritmo:

- Los nodos que representan un final de palabra tienen como enlace de fallo a la raiz.

Para aplicar esto tenemos las funciones:

```
1  // Aplica las primeras dos invariantes
2  Queue invariantes_Aho_Corasick(Diccionario);
3  //Enlaza los nodos
4  void enlazar_prefijo(Diccionario, Diccionario, char);
5  //Busca un enlace terminal si es que existe
6  void enlazar_terminal(Diccionario);
```

El motivo por el cual la primera función retorna una Queue es porque para poder aplicar el algoritmo sin encontrarnos punteros nulos, el algoritmo se debe aplicar a cada nodo por nivel, ya que cada enlace de fallo apunta a un nodo con una profundidad menor (o igual en el caso de la raíz).

Algoritmo de espaciado

Habiendo generado automata correctamente no es muy difícil poder localizar las palabras a espaciar ya que toda la estructura nos permite encontrarlas fácilemente.
Lo que que vamos a hacer es definir una estructura que represente un intervalo de números y conforme vayamos encontrando las palabras en la cadena entonces vamos a guardar el intervalo desde el inicio de la palabra hasta el final en una lista y después vamos a utilizar dicha lista para generar la oración espaciada.

```
typedef struct {
  int inicio;
  int final;
} _Intervalo;

typedef _Intervalo *Intervalo;
```

Para poder identificar a las palabras que comienzan primero y las que son más grandes, vamos a establecer una relación de prioridad entre los intervalos:

- Un intervalo es más prioritario que otro si comienza antes
- Si los dos intervalos comienzan en la misma posición entonces es más prioritario el que sea de mayor longitud

Una vez definido los intervalos y las prioridades, solo queda recorrer la oración visitando los nodos correspondientes del trie en el proceso.
Recorremos la oración y por cada letra nos fijamos si el nodo actual tiene un hijo asociado a esa letra. Si el nodo no tiene un hijo asociado, entonces pasamos al enlace de fallo y repetimos hasta encontrar un nodo que tenga un hijo asociado o hasta llegar a la raíz. Cuando sucede alguno de estos dos casos, leemos la siguiente letra y repetimos el proceso.
Si alguno de los nodos encontrados es de final de palabra o un nodo con enlace terminal entonces significa que la ultima letra leida es el final de una posible palabra de la oración y por lo tanto creamos el intervalo correspondiente a la palabra. Para poder encontrar en que posicion de la cadena se encuentra el inicio de la palabra solo basta en tomar la profundidad del nodo si el nodo es final de palabra ó tomar la profundidad del enlae terminal en el otro caso. La profundidad en este caso resulta ser la longitud de la palabra. La posición actual menos la longitud de la palabra más 1 nos da como resultado la posición del inicio de la palabra.
Con el intervalo creado, solo queda guardar la palabra en la lista eliminando los intervalos que tengan menor prioridad.

```
// Procesa la oracion y retorna una lista con los intervalos de
  las palabras validas
SList procesar_oracion(Parser, char *);
```

Habiendo echo todo lo descripto, solo queda recorrer la lista desde el primer elemento ingresado hasta el último para poder recuperar las palabras de la oración buscada.