Práctica 1

Construir un analizador léxico que devuelva los componentes léxicos que aparecen en el programa wilcoxon.py, codificado en Python.

Orientación

Los siguientes comentarios tan sólo pretenden servir como sugerencia para el diseño y desarrollo del analizador que se pide. Plantean una forma de resolución un tanto pedestre, pero suficiente, aunque admiten un amplio margen de mejora que será valorado.

Podemos descomponer el diseño en un conjunto de cinco ficheros de código: 1) un fichero de definiciones accesibles desde los otros tres ficheros; 2) un fichero que albergará el analizador léxico propiamente dicho; 3) un fichero donde implementaremos la tabla de símbolos y sus funciones de acceso; 4) un fichero que albergará el sistema de entrada; 5) un fichero que albergará el código de gestión de errores. Adicionalmente, podemos utilizar un fichero que invocará al analizador léxico para solicitar el siguiente componente léxico del código fuente, mostrándolo por pantalla para poder realizar la evaluación de la práctica.

Comentaremos brevemente una posible implementación para estos ficheros:

FICHERO DE DEFINICIONES

Puede contener las definiciones de los componentes léxicos del lenguaje Python utilizados en wilcoxon.py. Es suficiente con identificar cada componente léxico con un número entero, que el analizador léxico devolverá cada vez que sea invocado. Además, podemos definir aquí la tabla de símbolos como un vector de estructuras que permite almacenar cada uno de los componentes léxicos, y el lexema correspondiente en aquellos casos en los que sea necesario.

Un ejemplo de definición de componentes léxicos de Python, en C, podría ser la siguiente:

```
#define IMPORT 273

#define RETURN 274

        ...
```

FICHERO CON EL ANALIZADOR LÉXICO

Puede contener la función más relevante de esta práctica, cuya ejecución supone devolver el siguiente componente léxico del código fuente cada vez que es invocada. Esta función a su vez puede invocar al sistema de entrada para obtener el siguiente carácter del código fuente. Cada vez que obtiene un carácter ejecuta una estructura de autómatas que le permite identificar a qué patrón responde el flujo de entrada, y así devolver su componente léxico correspondiente. Por ejemplo, una vez que encuentra un paréntesis después de leer los caracteres ‘w’,’i’,’l’,’c’,’o’,’x’,’o’,’n’,’_’,’t’,’e’,’s’,’t’, el analizador debe comprobar que el lexema ‘wilcoxon_test’ no es una palabra reservada de Python, y después ha de incorporarlo a la tabla de símbolos como un identificador. Para terminar, ha de devolver el componente léxico IDENTIFICADOR, y guardar de alguna forma la cadena ‘wilcoxon_test’.

Una posible estructura muy sencilla para realizar el analizador podría ser la siguiente:

```
int seguinte_comp_lexico()

{

  while(!erro){

    switch (estado){

    case 0:

       c = seguinte_caracter();

       if (c == ' ' || c == '\t');

       else if (isapha(c) || c == '_')

          …

       else if (isdigit(c)) {

          …

    case 1:

    …

  }

}
```

donde la estructura `switch` implementa un autómata en el que las variables ‘estado’ y ‘c’ permiten simular la ejecución de los distintos AFD que identifican a cada uno de los componentes léxicos del lenguaje. El salto entre los distintos AFD se puede realizar mediante una función fallo() que selecciona el estado del siguiente AFD a simular cuando un AFD no termina su ejecución en un estado de aceptación.

FICHERO CON LA TABLA DE SÍMBOLOS

Fundamentalmente ha de contener dos funciones imprescindibles: 1) una función de búsqueda que permita recorrer la tabla de símbolos para no replicar innecesariamente el mismo componente léxico; 2) una función de inserción que permita almacenar un nuevo componente léxico encontrado.

FICHERO CON EL SISTEMA DE ENTRADA

Desde aquí se accede al fichero wilcoxon.py para analizarlo como un flujo de caracteres. El sistema de entrada ha de devolver el siguiente carácter del código fuente cada vez que es invocado, y ha de permitir devolver caracteres al flujo de entrada si es necesario. Podemos, por

tanto, considerar para ello el diseño de dos funciones.

Algunas funciones en C que pueden resultar útiles para el desarrollo de esta práctica:

`isalpha(), isdigit(), isalnum(), atoi(), atof(), getc(), strcpy()`