PRÁCTICA FLEX

Analizador HTML

Descripción breve

Mediante flex, hemos ideado un programa que analiza un HTML y muestra diversos parámetros: colores, etiquetas y posibles errores de formato.

Andrés José García Macías, José Mª Poblador Márquez

4. GIIADE, MC

1. Introducción

`flex` permite, en el enfoque de esta práctica, realizar acciones sobre código C++ cuando detecta expresiones regulares. Entonces, nuestra propuesta de programa pasa por un analizador de archivos HTML, que es capaz de, tras leer un documento `.htm(l)` y analizar sus líneas, deducir el número de etiquetas usadas y de colores para las distintas secciones que lo requieren, así como de listar cada uno de ellos.

En el caso de las etiquetas, comprueba además que cada una está debidamente cerrada (en HTML, la mayoría de secciones incorporan una etiqueta de apertura `<label>` y otra de cierre `</label>`), y muestra frecuencia de aparición relativa de manera visual con una barra.

2. Descripción de uso

El programa ejecuta de la siguiente manera:

- En la llamada, se pasa como argumento de llamada un fichero `.htm(l)` (que debe estar en la misma carpeta que el programa). También sirve su *path* absoluto.
- El programa va almacenando lo que corresponde a las reglas definidas: colores (sentencias tipo `#hex` o `rgb(x,y,z)`) y etiquetas (`<label></label>`). Almacena tanto la frecuencia como el valor detectado en sí.
- A la salida, muestra el número de colores y etiquetas detectadas, lista cada uno de los colores y las etiquetas. En este último caso, comprueba el número de etiquetas iniciales y finales: de no ser iguales, alguna sección está mal cerrada y así lo indica (`ABIERTO`/`cerrado`). Termina mostrando su barra de frecuencia relativa.

Para ejecutar el programa, basta llamar al script `./init.sh` seguido del fichero que se quiere analizar. El script incorpora tanto la llamada al precompilador `flex++` como al compilador de GNU y la llamada al programa. De no existir el archivo indicado o estar dañado, el programa devuelve un error y aborta.

Con el programa se incluyen dos archivos para probar su funcionamiento: un pequeño archivo de prueba llamado `prueba.html` y otro algo más grande extraído de la web del Developer Student Club llamado `dsc.htm`.

3. Análisis del código

Todo archivo de `flex` (`.l`) tiene 3 secciones. Comenzamos con la de declaraciones.

3.1. Sección de declaraciones

```
%option noyywrap

/*----- Sección de declaraciones ----------------*/
%{
#include <iostream>
#include <fstream>
#include <list>
#include <map>
```

```
using namespace std;

//Apertura de los ficheros .html
ifstream fichero;

//Declaración de variables
list<string> colores;                 //Lista de colores (#hex o rgb(..,..,..))
map<string, int> elementos;           //Map de etiquetas (finales: </..>), con frecuen
cia
map<string, int> elementosIniciales;  //Ídem con etiquetas iniciales (<..>)
int hashtagCount;                     //Contador de colores
int labelCount;                       //Contador de etiquetas

//Declaración de la función de escritura de datos
//Parámetros: hashtagCount labelCount
void escribir_datos (int dato1, int dato2);

//Declaración de la barra de carga
string barra_carga(int valor, int total);
%}
```

Para almacenar la frecuencia de colores y etiquetas, basta con dos enteros. Los colores pueden almacenarse en una lista STL; para las etiquetas nos hemos decantado por un diccionario al permitir almacenar pares clave-valor, donde la clave es la etiqueta y el valor su número de apariciones. Usamos uno para las etiquetas iniciales y otro para las finales, a modo de comparación. Así mismo, declaramos la función que escribirá los datos y la barra.

Pasamos ahora a describir las reglas.

3.2. Sección de reglas

```
/*----- Sección de reglas ----------------*/
%%
. {}
\<[a-zA-Z0-9]+ {
  string word(yytext);
  word = word.substr(1, word.length() - 1);

  if (elementosIniciales.find(word) == elementosIniciales.end()) {
    elementosIniciales.insert(pair<string,int>(word,1));
  }
  else {
    auto it = elementosIniciales.find(word);
    it->second += 1;
  }
}
\<\/[^\>\n]+\> {
```

```
  labelCount++;
  string word(yytext);
  word = word.substr(2, word.length() - 3);

  if (elementos.find(word) == elementos.end()) {
    elementos.insert(pair<string,int>(word,1));
  }
  else {
    auto it = elementos.find(word);
    it->second += 1;
  }
}
(#[0-9a-f]{6}|rgb[\(].*\)) {
    hashtagCount++; colores.push_back(yytext);
}

%%
```

La regla superior suprime el ECHO por defecto de `flex`. La primera regla es la que encuentra las etiquetas iniciales, que tienen la forma `<label>` o `<label [parameters]>`. Al encontrarlas, almacena la etiqueta (suprimiendo el < inicial y los demás caracteres tras ella) en el map de elementos iniciales, o, si ya existiese, aumentaría en 1 su frecuencia.

La segunda regla usa la misma mecánica que la primera para guardar las etiquetas finales. Esta vez, busca aquellas que siguen el patrón `</label>`. Además, esta regla es la que hace aumentar el contador de etiquetas.

La tercera regla busca los colores. En HTML hay varias maneras de definir colores, pero nosotros nos hemos decantado por detectar valores hexadecimales de 6 dígitos (`#abcdef`) y ternas RGB (`rgb(x,y,z)`). Cuando las detecta, aumenta el contador de colores en 1 y guarda la cadena en la lista.

Terminamos con la sección de procedimientos.

3.3. Sección de procedimientos

```
/*----- Sección de procedimientos --------*/
int main(int argc, char *argv[])
{
  //Llamada al fichero
  if (argc == 2)
  {
    fichero.open(argv[1]);
    if (fichero.fail())
    {
      cout << "error de lectura" << endl;
      exit(1);
    }
```

```
  }
  else
    exit(1);

  //Inicialización de parámetros
  hashtagCount = labelCount = 0;
  yyFlexLexer flujo(&fichero, 0);
  flujo.yylex();
  escribir_datos(hashtagCount, labelCount);

  return 0;
}

//Implementación de escribir datos
void escribir_datos (int dato1, int dato2)
{
  cout << "Num_colores = " << dato1 << endl;
  cout << "Num_etiquetas = " << dato2 << endl;

  //Itera la lista de colores
  cout << "\nCOLORES USADOS\n";
  list<string>::iterator it = colores.begin();
  for (; it != colores.end(); it++)
    cout << *it << endl;

  //Itera la lista de elementos y compara las ocurrencias de etiquetas finales e inic
iales
  //Si difieren, muestra en pantalla "ABIERTO"
  cout << "\nELEMENTOS USADOS\n";
  map<string, int>::iterator ite = elementos.begin();

  for (; ite != elementos.end(); ite++)
  {
    bool valoresIguales = false;
    if (elementosIniciales.find(ite->first) != elementosIniciales.end())
    {
      valoresIguales = (elementosIniciales.find(ite->first)->second != ite->second);
    }
    string estaCerrado = valoresIguales ? "ABIERTO" : "cerrado";
    cout << ite->first << '\t' << ite->second << '\t' << elementosIniciales.find(ite-
>first)->second << '\t'
         << estaCerrado << '\t' << barra_carga(ite->second, labelCount) << endl;
  }
}

//Implementación de la barra de carga
```

```
string barra_carga(int valor, int total) {
  string barra = "";
  float porcentaje = ((valor * 1.0)/(total * 1.0)) * 100;
  for (int i = 0; i < porcentaje / 5; i++)
    barra += "";

  return barra;
}
```

En esta sección, básicamente se inicializan los parámetros y se ordena a `lex` la ejecución de las reglas. Tras ello, se escriben los datos obtenidos en pantalla.

En la sección se incluye la implementación de dicha función y la de la barra de carga.

4. Ejemplos de ejecución

Ejemplos de ejecución con ambos ficheros: prueba.html y dsc.html