MC

MODELOS DE COMPUTACIÓN

Práctica 2

Autor: Sergio Campoy Maldonado

ESCUELA TÉCNICA SUPERIOR DE INGENIERÍAS INFORMÁTICA Y DE TELECOMUNICACIÓN

—

Curso 2020 - 2021

Índice

1. Descripción del problema

Con esta práctica, he intentado resolver uno de los problemas que surge de trabajar en grupo en un proyecto de c++: La discrepancia de estilo en el código.

Ya que la gente tiene distintas preferencias, el código puede acabar siendo un desastre: unas secciones indentadas con tabulador, otras con 3 espacios y otras con 2; algunas llaves que abren una función están al comienzo de la línea, otras al final, etc. Los siguientes códigos son un ejemplo de a lo que me refiero.

```
for (int i=0; i<10; i++) {
    if (i%2) {
        cout<<"Es par!"<<endl;
    } else {
        cout<<"Es impar"<<endl;
    }
}
```

```
for (int i = 0; i < 10; i ++ )
{
  if (i % 2)
  {
    cout << "Es par!" << endl;
  }
  else
  {
    cout << "Es impar" << endl;
  }
}
```

El objetivo que me he propuesto para la práctica es crear un programa de *flex* que, dados dos códigos de c++ iguales con distinto estilo, genere como resultado un mismo programa.

```
for (int i = 0; i < 10; i ++ ) {
    if (i % 2) {
        cout << "Es par!" << endl;
    } else {
        cout << "Es impar" << endl;
    }
}
```

2. Descripción de la solución

La solución compone de *4 ficheros flex*, un *script de ejecución* y un *Makefile*. Cada uno de los *programas de flex* tiene un objetivo: el primero añade espacios para que no haya operadores pegados con palabras o números, el segundo "comprime" todos los tabuladores, saltos de línea y espacios en un solo espacio, el tercero añade los saltos de línea y el cuarto tabula para obtener el resultado final.

2.1. Uso

1. Primero hay que compilar los ficheros, para lo que basta con ejecutar `make`.
2. Luego hay que añadir el fichero que se quiere convertir a la carpeta `ejemplos` y se le cambia la terminación a `.cc`
3. Una vez añadido, ejecutar el script con el nombre del fichero (sin la terminación) y el archivo resultante se genera en `resultado`. Este paso solo es necesario si se ha añadido un fichero extra a ejemplos, ya que el *makefile* llama al script con los ejemplos que proporciono.

2.2. Carpetas

La estructura de carpetas es la siguiente:

- `ejemplos` - Carpeta con con archivos `.cc` de ejemplo. Aquí se tiene que poner el archivo que se quiere convertir.
- `flex` - Carpeta con los archivos de flex.
- `src` - Aquí se guardan los archivos `.c` que generan los archivos `.flex`.
- `bin` - En esta carpeta se generan los ejecutables.
- `tmp` - En esta carpeta se guardan los archivos intermedios. No es necesario guardarlos, pero son útiles para comprobar qué hace el programa en cada paso.
- `resutado` - Aquí se guarda el código resutado de ejecutar el programa sobre un archivo de la carpeta `ejemplo`.

2.3. Ficheros

2.3.1. Makefile

El makefile es bastante sencillo. Tiene dos reglas `.PHONY`: `clean` y `all`, que se encargan de borrar los archivos temporales (`src, bin, tmp y resutado`); y de compilar los ficheros y ejecutar el script respectivamente.

También tiene dos reglas por cada fichero `.l` para generar el `.c` y el binario correspondientes.

```
src/primero.c : flex/primero.l
    flex -o src/primero.c flex/primero.l

bin/primero : src/primero.c
    gcc -o bin/primero src/primero.c

```

2.3.2. script.sh

El script recibe un argumento, que es el nombre del fichero a convertir sin el path ni la terminación (`ejemplos/ejemplo.cc` sería `ejemplo`) y el script ejecuta los cuatro binarios en orden sobre el fichero. Los pasos intermedios los guarda en `tmp` y el final en `resultado`.

```
#!/bin/bash

bin/primero < ejemplos/$1.cc > tmp/$11.cc
bin/segundo < tmp/$11.cc > tmp/$12.cc
bin/tercero < tmp/$12.cc > tmp/$13.cc
bin/cuarto  < tmp/$13.cc > resultado/$1.cc
```

2.3.3. primero.l

El objetivo del primer archivo es añadir espacios para evitar que haya operadores "pegados" con palabras. Al principio declaro variables relacionadas con los operadores y las llaves. Al final las importantes son `llave`, que detecta cualquier llave o corchete, y `ops`, que detecta cualquiera de los operadores de c++.

También declaro varias condiciones de inicio. El objetivo de estas es que no se modifique nada que haya en los comentarios, entre comillas o en los includes. Para ello, al leer unas comillas dobles, simples, el inicio de un comentario o #include hago `BEGIN(condicion)` y después al leer el final hago `BEGIN(INITIAL)` para volver al estado inicial.

2.3.4. segundo.l

El segundo archivo "comprime" los espacios en blanco y reduce el código a una línea (salvo que haya comentarios). Gracias a esto consigo homogeneizar todas las diferencias de espaciado (espacios vs. tabuladores, que haya más o menos indentación, que haya uno o más espacios entre los operadores, etc).

La carga principal de este fichero la tiene la primera regla, que convierte cualquier espacio, tabulador o espacio en uno solo (`[\n\t\ ]* { printf(" "); }`).

El resto de reglas sirven para dos cosas: que no se modifiquen los espacios de los comentarios y para diferenciar los comentarios que están en una linea propia de los que están detrás de código al final de una linea. En esencia funcionan de la misma manera que las condiciones iniciales del primer archivo, pero también añaden espacios o saltos de línea en función del caso.

2.3.5. tercero.l

El tercero se encarga de añadir los saltos de línea al programa. El primer grupo de reglas añade un salto después de leer la cadena seguida de un espacio salvo la regla de #include, que añade el salto antes.

El segundo grupo de reglas consigue que, si el comentario aparece al final de la línea, no se quite de ahí. Finalmente en el tercer grupo hay dos casos especiales del punto y coma donde no se van a añadir saltos: los dos que aparecen dentro de un bucle for y el que cierra una clase.

La regla del for es un poco más compleja que las demás, por lo que la voy a explicar:

- `"for ("` La cadena "for (" literal
- `[^\;]+` Cualquier caracter 1 o más veces que no sea ";". Acepta caracteres hasta llegar a un punto y coma
- `;` Un punto y coma literal
- `{2}` La expresión regular anterior 2 veces. Como hay paréntesis, se refiere a `[^\;]+;`
- `[^\)]+` Cualquier caracter 1 o más veces que no sea ")". Acepta caracteres hasta llegar a un paréntesis que cierra
- `")"` Un paréntesis que cierra literal

Una vez detectado el for, simplemente lo imprime de nuevo para no modificar su contenido.

2.3.6. cuarto.l

Finalmente el cuarto fichero añade los tabuladores al código. Para este programa he necesitado utilizar una variable global (`ntab`) inicializada a 0 para contar cuantos tabuladores tengo que poner y una función auxiliar (`void tabula()`) para imprimir los tabuladores.

En cuanto a las reglas, la principal es la primera, que añade a todo \n los tabuladores que hagan falta. Para que funcione bien son necesarias las dos siguientes, que aumentan o disminuyen `ntab` al encontrar una llave que abre o una que cierra.

Las tres siguientes son por preferencia personal, me gusta que `private, public y protected` esten menos indentadas que el resto del código. La última regla simplemente elimina unos espacios que sobran de algunas reglas del tercer programa.