Prácticas Modelos de Computación
Indentador código C/C++

Rubén Morales Pérez

4 de diciembre de 2016

1. Introducción

A pesar de que en los inicios de los cursos de programación se explica la importancia de tabular el código que se escribe no todo el mundo lo hace. Todos somos conscientes de lo difícil que es leer código cuando no está bien tabulado. Poner en la misma línea un if y su acción o poner la llave { en una línea diferente del último paréntesis, por estética o ahorro de espacios son algunos de estos malos vicios que una persona coge al escribir código.
Poner demasiadas líneas seguidas sin líneas en blanco puede resultar tan molesto como leer un texto largo en un solo párrafo o leer una frase sin ningún signo de puntuación Para los más puristas incluso no tener el mismo criterio para poner espacios (poner espacio detrás de una coma, no ponerlo, ponerlo a veces antes) puede resultar molesto cuando quieres leer código limpio y bien hecho.
Lo ideal sería que todo el código estuviese bien tabulado, para facilitar esta tarea se ha creado un tabulador automático para código C/C++. Dicho programa está hecho con LEX, para construir un analizador léxico con el que obtener nuestro objetivo.

2. Declaraciones

Incluiremos algunas bibliotecas necesarias para el programa

```
#include <stdio.h>
#include <stdlib.h>     /* atoi */
#include <ctype.h>      /* tolower */
#include <sys/wait.h>
typedef enum { false, true } bool;
```

2.1. Variables

```
long int numIndex   = 0;      // Tabulaciones necesarias
int  esBloque       = 0;      // (Bloque)

bool inicioLinea    = true;
bool insertadaLinea = false;
bool lineaProcesada = false;
bool anteriorLlave  = false;  // Llave '}' solamente
bool salidaFichero  = false;  // Solo se modifica 0/1 veces
```

- *numIndex* es el número de tabulaciones necesarias en cada instante.
- *esBloque* nos dice si estamos dentro de un paréntesis o no, esto es útil ya que después de un ; siempre irá un $\backslash n$, salvo que esté dentro de un *for* o de una cadena de texto.
- *inicioLinea* nos dice si estamos en el inicio de una línea, es necesario para saber si tenemos que meter *numIndex* $\backslash t$ o no.
- *insertadaLinea* nos dice si se ha impreso justo antes un $\backslash n$, es necesario ya que pueden darse situaciones en las que sea necesario insertar un $\backslash n$ manual, con esta variable nos aseguramos de no insertar varios saltos de línea redundantes.
- *lineaProcesada*, tras un { que no sea texto siempre irá un salto de línea y un aumento de *numIndex*, salvo que ya hayamos encontrado una estructura de control y aumentado *numIndex* anteriormente.
- *anteriorLlave* nos dice si el último carácter es una llave }. Esto nos sirve para poner un salto de línea extra en estructuras de control largas pero si aparecen varias llaves seguidas no tengan líneas en blanco en medio.
- *salidaFichero* nos indica si la salida por pantalla es por la salida estándar o no.

2.2. Funciones

```
1 void escribir();
```

Nos permite imprimir el texto asociado a la expresión regular reconocida teniendo en cuenta si es necesario tabular o no.

```
1 void escribirCadena(char *cadena, int n);
```

Imprime los n primeros caracteres de la cadena indicada sin tener en cuenta tabulaciones.

```
1 void bloque();
```

Nos ayuda a identificar si la cadena deseada está entre paréntesis, necesario para no imprimir un salto de línea tras el ; de un bucle *for*.

```
1 int ultimoParentesisDerecho();
```

Nos devuelve el último paréntesis derecho que aparece en *yytext*, esto nos ayuda a separar una estructura de control de la sentencia cuando no se separan por un $\backslash n$, ejemplo:

```
1 if(obsesionPorElEspacio) noTabularCorrectamente();
```

```
1 bool hayComentario(int *lineaComent);
```

Devuelve si hay algún comentario en *yytext* y devuelve en *lineaComent* el inicio de dicho comentario.

```
  void depurar(char * cadena);
```

Sentencia que nos indica la expresión regular por la que ha sido reconocido yytext.

2.3. Expresiones regulares

2.3.1. Texto

```
  /* Texto */
espacio                 (\ |\t)
blanco                  ({espacio}|\n)
caracter                ('(.|\\n|\\t|\\b)')
cadenaTexto             ((\"([^(\")]|"\"")+\")|{caracter})
```

Expresiones rutinarias, quizá la más interesante sea *cadenaTexto*, ya que es necesario incluir la opción de incluir unas comillas dentro de una cadena de texto.

2.3.2. Estructuras de control

```
  /* Estructuras de control */
obj                     (struct|class|union)
control                 (if|else|for|while|switch|case)
excepcion               (try|catch)
clase                   (public|private)
main                    (int{blanco}+main)

tipomas                 (({espacio}*)({obj}|{control}|{excepcion}|{clase}|{main}))
```

tipomas y *tipomenos* son las que se utilizan en el programa, identifican los momentos en los que hay que aumentar o disminuir el número de tabulaciones. El resto son palabras reservadas que nos ayudan con la tabulación.

2.3.3. Comentarios

```
  /* Comentarios */
comentarioLinea         \/\/.*\n
comentarioLargo         ("/*"([^*]|(\*+[^\/]))*\*\/)
```

comentarioLargo es una expresión regular interesante.

2.3.4. Separación con espacios

```
  /* Separaciones con espacios */
tipoDoble               (=|==|\+=|-=|\*=|\/=|\+|-|\*)
tipoVacio               ("++"|"--"|\(|\))
parentesis              (\([^;]+\))
parentesisFor           (\([^;]*;[^;]*;[^;]*\))

espacioDoble            ({espacio}*{tipoDoble}{espacio}*)
espacioVacio            ({espacio}*{tipoVacio}{espacio}*)

llaveFea                ({blanco}*\)({comentario}|{blanco})*\{)
puntoComa               ({espacio}*;{espacio}*{comentarioLinea}?)
```

Las primeras son expresiones auxiliares, tienen comportamiento asociado los 5 últimos. *espacioDoble* y *espacioVacio* identifica las sentencias que tienen obligatoriamente un espacio a cada lado o las que no deben tener ninguno y se traga los espacios a los lados para tener un formato homogéneo. Anteriormente también se creó *espacioIzquierda* para intentar poner un espacio justo antes del paréntesis izquierdo en una estructura de control o llamando a una función, pero se optó por otra solución.
llaveFea identifica la sintaxis del tipo

```
1  if(llaveEnOtraLinea) // Comentario opcional (o comentario largo)
2  {
3    sentencias...
4  }
```

que será transformada en

```
1  if(llaveEnOtraLinea){ // Comentario opcional (o comentario largo)
2    sentencias...
3  }
```

puntoComa crea un formato homogéneo para los comentarios al lado del código, también asegura el salto de línea después de un ';', para evitar varias sentencias en la misma línea.

controlSinComa nos identifica las sentencias de control de una línea que no requieren ir entre llaves. Para no tener que volver a pasar *yylex*() en el mismo programa invocamos dos esclavos que nos harán trabajo por separado.

3. Reglas

- {comentario} Deja el comentario igual asegurando el salto de línea al final.
- {cadenaTexto} Imprime tal cuál la cadena de texto teniendo en cuenta las restricciones de tabulación.
- (\ )+ Deja un código más homogeneo, aunque hay situaciones en las que puede preferirse quitar esta regla.
- (\t)+ Igual que el anterior
- {*espacio*}$*$, {*espacio*}$*$ Deja una coma y un espacio inmediatamente después.
- {puntoComa} Deja un ; y un espacio, tabulación o salto de línea en función de en qué posición del código se encuentre dicho carácter. Por ejemplo, si está en un bucle for solamente deja un espacio para que sea más legible.
- {espacioDoble} Deja coma a izquierda y derecha del carácter identificado.
- {espacioVacio} Elimina espacios y tabulaciones alrededor del carácter deseado.
- {llaveFea} Arregla la situación descrita anteriormente.
- {controlSinLinea} Modifica líneas donde falten saltos de línea a mitad, normalmente usadas por ahorro de espacio.

- {tipomas} Suma una tabulación a las necesarias para garantizar la indentación del código.
- {tipomenoss} Resta una tabulación.
- $\backslash n$ Imprime un salto de línea salvo que haya sido ya impreso como consecuencia de otra regla.
- . Imprime el carácter respetando tabulaciones.

4. Procedimientos

```
int main (int argc, char *argv[]){
  if (argc == 2 || argc == 3){
    yyin = fopen (argv[1], "rt");

    if (yyin == NULL){
      printf ("El fichero %s no se puede abrir para lectura.\n", argv[1]);
      exit (-1);
    }
    if (argc == 3){
       //fichSalida = argv[2];
       //fp = fopen(fichSalida, "w");
       yyout = fopen(argv[2], "w");

       salidaFichero = true;
    }
  }
  else return 1;

  yylex ();
  return 0;
}
```

5. Ejemplos

Veamos como se comporta el programa con ejemplos reales.

```
if (control)  x=x+y;  for (int i  =0; i<n ;++i ) x+=i;
```

Listing 1: Ejemplo1

```
if (control){
   x = x + y;
}
for (int i = 0; i<n; ++i){
   x += i;
}
```

Listing 2: Ejemplo1Salida

Probamos con un main

```
int main (int argc, char *argv[])
{
  if (argc == 2)
```

```
  {
yyin = fopen (argv[1], "rt");

    if (yyin == NULL)
    {
        printf ("El fichero %s no se puede abrir\n", argv[1]);
      exit (-1);
    }
  }
  else{
      yyin = stdin;
  }
  nc     =  np = nl = 0;
  yylex (   );
  escribir_datos(nc,np,     nl);

  return 0;
}
```

Listing 3: Ejemplo2

```
int main (int argc, char * argv[]){
  if (argc == 2){
    yyin = fopen (argv[1], "rt");
    if (yyin == NULL){
      printf ("El fichero %s no se puede abrir\n", argv[1]);
      exit ( - 1);
    }
  }
  else{
    yyin = stdin;
  }
  nc = np = nl = 0;
  yylex ();
  escribir_datos (nc, np, nl);
  return 0;
}
```

Listing 4: Ejemplo2Salida

6. Sugerencia para LEX

La visualización de las expresiones regulares no es nada intuitiva, cuando escribes alguna con cierta complicación sin ningún espacio pierdes agilidad al leerla. Una forma de solucionar esto podría ser tener un carácter especial para el espacio (parece que puede haber fallos si no se escapan los espacios, esto lo solucionaría) y que te dejase separar las expresiones regulares con los espacios deseados.
Ejemplo:

```
cadenaTexto             (\"([^(\")]|"\"")+\")|{caracter}
```

podría pensarse más rápido si LEX permitiese hacer

```
cadenaTexto             (\"  ( [^(\")] | "\"" )+  \")    |    {caracter}
```