Parsers.

Programación I – Laboratorio I.
Tecnicatura Superior en Programación.
UTN-FRA

Autores: *Esp. Ing. Ernesto Gigliotti*

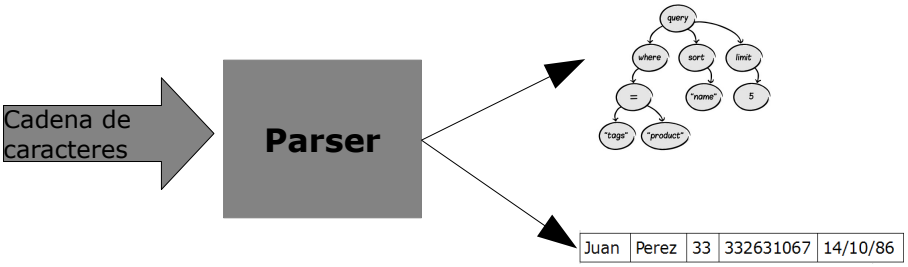
Revisores: *Esp. Lic. Mauricio Dávila*

Versión : 1.3

Índice de contenido

1 ¿Qué es un parser?

Un *parser* o analizador sintáctico, se encarga de analizar una cadena de caracteres conforme a ciertas reglas. Generalmente un *parser* posee un *input* de datos (binarios o texto) el cual utiliza para construir una estructura de datos con cierta complejidad (puede ser tan complejo como un árbol, el cual jerarquiza la información, o tan simple como un *array* de palabras)



1.1 Utilización de un *parser*

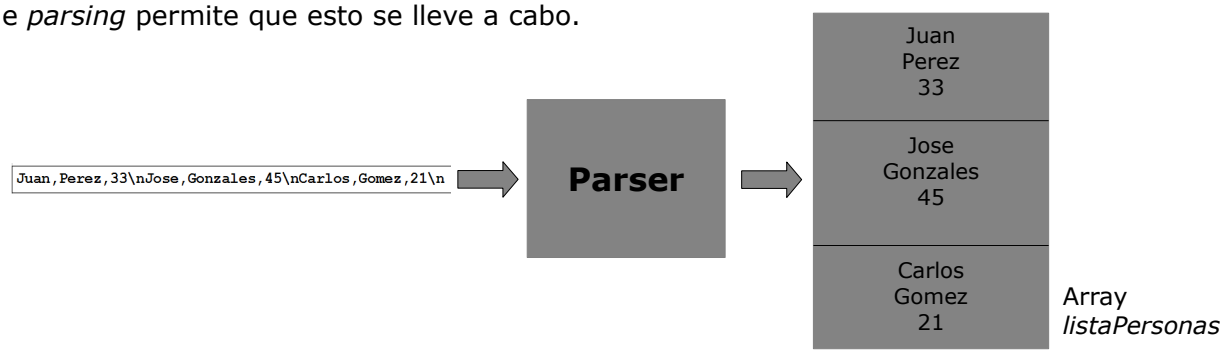
Haremos uso de este tipo de analizadores cuando necesitemos interpretar información que se encuentra almacenada bajo ciertas reglas o "formato". Por ejemplo, si se quiere representar una persona, y la regla nos dice que los datos de nombre, apellido y edad están separados por un carácter especial "," (coma) y que cada persona estará separada de la siguiente por medio de otro carácter especial "\n" (enter), nos podemos encontrar con una cadena de caracteres como la siguiente:

```
Juan,Perez,33\nJose,Gonzales,45\nCarlos,Gomez,21\n
```

La cual puede encontrarse escrita en un archivo o provenir de otro medio. Sin embargo, si debemos representar a la entidad "Persona" en nuestro programa, deberemos definir una estructura de datos como la siguiente:

```
struct S_Persona
{
    char nombre[32];
    char apellido[32];
    int edad;
};
typedef struct S_Persona Persona;
Persona listaPersonas[3];
```

Como se observa, nuestro programa tendrá definido en memoria un *array* de estructuras "Persona" pero para poder cargar los campos de cada uno de los ítems de este *array*, necesitaremos leer el archivo descripto anteriormente, y mediante algún algoritmo, cargar los campos de las variables "Persona" con el contenido de dicho archivo. Es aquí cuando el proceso de *parsing* permite que esto se lleve a cabo.



1.2 Creación de un *parser*

Para nuestro ejemplo, un *parser* no será mas que una función, la cual abre un archivo y lo lee (obtenemos de esta forma el *input* del parser), y llena un *array* pasado como parámetro con la información leída, como se indica en el diagrama anterior.

Mediante la función ***fscanf,*** podremos leer de un archivo una cadena de caracteres mientras se cumpla una regla descripta, que deberá ser pasada a la función como parámetro.

La función analizará la cadena y continuará leyendo mientras se cumpla la regla que se le pasó. En dicha regla podremos incluir indicaciones, para que la función almacene los datos encontrados en una variable también pasada como argumento.

Ejemplo:

Tenemos un archivo con el siguiente contenido:

```
1,2,3,4\n5,6,7,8\n9,10,11,12\n
```

Suponiendo que cada bloque de información idéntica está separado por "\n", tenemos 3 bloques de información para leer del archivo. El *parser* para este formato es el siguiente:

```
int main(void)
{
    FILE *pFile;
    int r;
    int a,b,c,d;

    pFile = fopen("data.csv","r");
    if(pFile == NULL)
    {
        printf("El archivo no existe");
        exit(EXIT_FAILURE);
    }
    do
    {
        r = fscanf(pFile,"%d,%d,%d,%d\n",&a,&b,&c,&d);
        if(r==4){
            printf("Lei %d %d %d %d\n",a,b,c,d);
        }
        else
            break;

    }while(!feof(pFile));

    fclose(pFile);
    exit(EXIT_SUCCESS);
}
```

Como se observa en el código, el programa abre el archivo "data.csv" en modo lectura, de esta forma obtenemos el puntero FILE* llamado pFile. La línea de código que lee datos del archivo se detalla a continuación:

```
r = fscanf(pFile,"%d,%d,%d,%d\n",&a,&b,&c,&d);
```

La función recibe como primer argumento el puntero al archivo (pFile), luego recibe la regla a aplicar sobre los datos leídos, y luego recibe una cantidad indefinida de punteros a las variables (a,b,c y d en este caso) en donde se almacenarán los números encontrados según lo especificado en la regla pasada como parámetro.La función retornará cuántas variables pudo cargar, (en este ejemplo 4 si pudo encontrar en la cadena leída todo lo descripto en la regla).

Analicemos ahora la regla utilizada para analizar la información en el archivo. El archivo está compuesto por el siguiente contenido:

```
1,2,3,4\n5,6,7,8\n9,10,11,12\n
```

En decir que posee datos, separados por coma, luego se encuentra un enter ("\n") para separar un bloque de datos del siguiente.

Deberemos escribir una regla que describa el "formato" de los datos, y luego ejecutar la función ***fscanf*** para leer en forma repetitiva los datos de cada bloque. Para ello, se analiza cómo es el formato de la información en el archivo para un solo bloque:

CAMPO1	,	CAMPO2	,	CAMPO3	,	CAMPO4	\n

Luego se analiza de qué tipo de dato es cada campo, en este ejemplo simple, son numéricos. Para escribir la regla, deberemos ir escribiendo lo mismo que nos vamos encontrando al leer la cadena, para indicar números utilizaremos **%d** (al igual que con la función ***scanf***), para indicar una palabra (sin espacios) utilizaremos **%s**. Si sabemos exactamente qué carácter o caracteres van a encontrarse en la cadena, los escribimos como parte de la regla. De esta forma, nuestra regla queda:

CAMPO1	,	CAMPO2	,	CAMPO3	,	CAMPO4	\n
%d	,	%d	,	%d	,	%d	\n

Pasamos dicha regla como una cadena a la función fscanf : `"%d,%d,%d,%d\n"`

Como especificamos en la regla 4 campos con "%d", deberemos pasar a la función 4 variables en donde escribirá el valor encontrado en la cadena para dichos 4 campos (las variables a,b,c y d).

La salida por consola del programa, suponiendo que la información en el archivo es la descripta anteriormente, será la siguiente:

```
Lei 1 2 3 4
Lei 5 6 7 8
Lei 9 10 11 12
```

Modifiquemos el ejemplo anterior suponiendo que la información en el archivo nos describe a personas y es la que se detalla a continuación:

```
1,Juan,Perez,33\n2,Jose,Gonzales,45\n3,Carlos,Gomez,21\n
```

El primer campo es un ID, el segundo un nombre, el tercero un apellido y el cuarto la edad.

Podemos entonces analizar el formato para una sola persona ya que luego se repetirá:

CAMPO1	,	CAMPO2	,	CAMPO3	,	CAMPO4	\n
%d	,	%s	,	%s	,	%d	\n
ID		*Nombre*		*Apellido*		*Edad*	

De esta manera determinamos la regla: `"%d,%s,%s,%d\n"`. Sin embargo, esta regla no funcionará, ya que "%s" implica una palabra, cualquier conjunto de caracteres menos el espacio. Esto quiere decir que la coma "," está incluida dentro del conjunto que se expresa con "%s", por lo tanto la función ***fscanf*** nos devolverá un 2, solo detectará dos campos, ya que interpretará el contenido de la siguiente manera:

CAMPO1	,	CAMPO2	,	CAMPO3	,	CAMPO4	\n
%d	,			%s			

Para solucionar este problema, deberíamos indicar un conjunto de letras en donde estén todas las letras, menos la coma "," ya que ésta es la que determinará que el campo termina al encontrar dicho carácter. Tendríamos el mismo problema con los campos numéricos, pero el "%d" representa el conjunto de caracteres del 0 al 9, por lo que la coma no esta allí.

CAMPO1	,	CAMPO2	,	CAMPO3	,	CAMPO4	\n
%d	,	Conjunto de letras menos la ","	,	Conjunto de letras menos la ","	,	%d	\n
ID		*Nombre*		*Apellido*		*Edad*	

Para indicar un conjunto de letras, podemos utilizar corchetes, indicando dentro un rango de caracteres, por ejemplo: "%[a-z]" o "%[A-Z]".

También podemos expresar un conjunto negado, es decir "cualquier carácter menos los que están en este conjunto", para ello indicamos el conjunto con un "^" delante, por ejemplo: "%[^,]" indica "cualquier carácter menos la coma".

Ahora podemos reescribir nuestra regla de la siguiente manera: `"%d,%[^,],%[^,],%d\n"`

CAMPO1	,	CAMPO2	,	CAMPO3	,	CAMPO4	\n
%d	,	%[^,]	,	%[^,]	,	%d	\n
ID		*Nombre*		*Apellido*		*Edad*	

```
char nombre[50];
char apellido[50];
int id,edad;
...
do
{
    r = fscanf(pFile,"%d,%[^,],%[^,],%d\n",&id,nombre,apellido,&edad);
    if(r==4){
        printf("Lei %d %s %s %d\n",id,nombre,apellido,edad);
```

Otra cosa que deberemos cambiar en el programa, es que las variables que se escribirán con %d son del tipo "int", mientras que las indicadas con %s o %[] deben ser una cadena de caracteres, como se muestra en la porción de código anterior.

Para finalizar, se recomienda leer todos los campos como cadena de caracteres, y en un paso posterior a su detección y almacenamiento en una variable, se deberán validar y convertir al tipo de dato necesario.

Si el útimo campo es vez de ser numérico es una o varias palabras, deberemos indicar un conjunto que sea "cualquier carácter menos el enter", de la siguiente forma:

CAMPO1	,	**CAMPO2**	,	**CAMPO3**	,	**CAMPO4**	**\n**
Conjunto de letras menos la ","	,	**Conjunto de letras menos la ","**	,	**Conjunto de letras menos la ","**	,	**Conjunto de letras menos el "\n"**	**\n**
%[^,]	,	**%[^,]**	,	**%[^,]**	,	**%[^\n]**	**\n**
cadena		*cadena*		*cadena*		*cadena*	

A continuación se muestra un programa en donde cada campo es obtenido como cadena:

```
int main(void)
{
    FILE *pFile;
    int r;
    char var1[50],var3[50],var2[50],var4[50];

    pFile = fopen("data.csv","r");
    if(pFile == NULL)
    {
        printf("El archivo no existe");
        exit(EXIT_FAILURE);
    }

    do
    {
        r = fscanf(pFile,"%[^,],%[^,],%[^,],%[^\n]\n",var1,var2,var3,var4);
        if(r==4)
            printf("Lei: %s %s %s %s\n",var1,var2,var3,var4);
        else
            break;

    }while(!feof(pFile));

    fclose(pFile);

    exit(EXIT_SUCCESS);
}
```

Para más información sobre las expresiones que pueden utilizarse con la función ***fscanf***, chequear la documentación de la misma: http://www.cplusplus.com/reference/cstdio/fscanf

1.3 Creación de estructuras de datos

Para que nuestro *parser* esté completo, deberemos cambiar el código anterior, para que en vez de solamente imprimir los datos almacenados en las variables (var1, var2, var3, y var4) por medio de la función **printf**, se almacene dicha información en campos de una estructura perteneciente a un *array*.

Supongamos que definimos la estructura Persona y un *array* de personas en nuestro programa. La idea de la función **parseData** es que le pasemos el nombre de un archivo, y el *array* sin inicializar, y nos devuelva el *array* cargado:

Ejemplo:

```
struct S_Persona
{
    int id;
    char nombre[32];
    char apellido[32];
    int edad;
};
typedef struct S_Persona Persona;

int parseData(char* fileName,Persona* arrayPersonas, int len);

int main(void)
{
    Persona personas[8];
    int r,i;

    r = parseData("data.csv",personas,8);
    for(i=0; i<r; i++)
        printf("id:%d nombre:%s apellido:%s edad:%d\n", personas[i].id,
                                                        personas[i].nombre,
                                                        personas[i].apellido,
                                                        personas[i].edad);
    exit(EXIT_SUCCESS);
}
```

Para ello utilizaremos el código anterior el cual pondremos dentro de una función, la cual tendrá el prototipo:

```
int parseData(char* fileName,Persona* arrayPersonas, int len);
```

Solo cambiaremos la porción del bloque “if” en donde se detectaba que se habían leído los campos y se imprimían, y en ese lugar se cargarán los datos en una posición del *array* recibido como argumento.

A continuación se detalla la función completa:

```
int parseData(char* fileName,Persona* arrayPersonas, int len)
{
    FILE *pFile;
    int r,i=0;
    char var1[50],var3[50],var2[50],var4[50];

    pFile = fopen(fileName,"r");
```

```
    if(pFile == NULL)
    {
        return -1;
    }

    do
    {
        r = fscanf(pFile,"%[^,],%[^,],%[^,],%[^\n]\n",var1,var2,var3,var4);
        if(r==4)
        {
            arrayPersonas[i].id = atoi(var1);
            strncpy(arrayPersonas[i].nombre,var2,sizeof(arrayPersonas[i].nombre));
            strncpy(arrayPersonas[i].apellido,var3,sizeof(arrayPersonas[i].apellido));
            arrayPersonas[i].edad = atoi(var4);
            i++;
        }
        else
            break;

    }while(!feof(pFile) && i<len);

    fclose(pFile);
    return i;
}
```

En este código, se inicializa la variable "i" en cero, la cual se utilizará como índice para ir cargando cada ítem del array, y se incrementará en cada iteración. Dentro del bloque "if" se cargan los campos del ítem en la posición "i" convirtiéndolos al tipo de dato que se necesita, leyéndolos de las variables auxiliares var1, var2, var3 y var4.

El *parser* terminará por 3 razones:

- El formato no es el correcto. En este caso *r != 4*
- La cantidad de personas es mayor que "len". En este caso se deja de cumplir *i<len*
- Se alcanzó el fin del archivo.

Y devolverá la cantidad de personas que se cargaron en el *array* (que coincide con el valor en la variable "i").

1.4 Parser con creación dinámica de datos

En el ejemplo anterior trabajamos con un *array* estático de estructuras, el mismo tenía un tamaño definido haciendo que debamos ocupar mucha memoria en variables que quizá no usaremos. Al trabajar con memoria dinámica, este problema está resulto, modificaremos el código anterior suponiendo que existe una biblioteca "Persona.c" y "Persona.h" la cual posee funciones para crear en forma dinámica una estructura del tipo Persona.

Las funciones que suponemos que existen son:

```
Persona* per_newPersona(void);
int per_setId(Persona* p,int id);
int per_setEdad(Persona* p,int edad);
int per_setNombre(Persona* p,char* nombre);
int per_setApellido(Persona* p,char* apellido);
```

De esta manera podemos modificar la función para que reciba un *array* de **punteros** del tipo Persona:

```
int parseData(char* fileName,Persona* arrayPersonas[], int len)
```

Por último modificamos el bloque "if" y creamos las variables solo cuando podemos leer los datos, y utilizamos las funciones descriptas para cargar los mismos.

```
r = fscanf(pFile,"%[^,],%[^,],%[^,],%[^\n]\n",var1,var2,var3,var4);
if(r==4)
{
    Persona* pAux = per_newPersona();
    per_setId(pAux,atoi(var1));
    per_setNombre(pAux,var2);
    per_setApellido(pAux,var3);
    per_setEdad(pAux,atoi(var4));
    arrayPersonas[i] = pAux;
    i++;
}
```