INGENIERÍA | DEPARTAMENTO
UNIVERSIDAD DE LOS ANDES | DE COMPUTACIÓN
MÉRIDA VENEZUELA | ESCUELA DE SISTEMAS

Compiladores
Analizador Léxico y Sintáctico
Explicación Caso de Estudio

Prof. Hilda Contreras
Grupo Nro. 2
Semestre B-2018

1 Flex

Flex es una herramienta para generar "scanners", o analizadores léxicos. Un scanner a veces llamado "tokenizador", es un programa el cual reconoce patrones léxicos en un texto. El programa generado lee desde un archivo o desde la entrada por defecto para la especificación de un analizador léxico. La descripción de este se encuentra en reglas formadas por pares de expresiones regulares y código en lenguaje `C`.

1.1 Instalación

Para sistemas basados en Debian ingrese el siguiente comando en su terminal:

```
usuario@pc:~$ sudo aptitude install flex
```

Verifique que la instalación se realizó con éxito, comprobando la versión de flex

```
usuario@pc:~$ flex --version
```

1.2 Estructura de un archivo Flex

Un archivo en Flex posee el siguiente formato dividido en tres secciones separadas por un doble caracter porcentaje '% %', estas son:

1.2.1 Sección de definiciones

La sección de definiciones contiene declaraciones de "nombres" los cuales son identificadores para expresiones regulares y también para las declaraciones de condiciones. Se pueden incluir en esta sección archivos cabeceras y prototipos de funciones a ser usados en el analizador léxico.

1.2.2 Sección de reglas

La Sección de reglas contiene propiamente reglas de la forma:

```
Patrón Acción
```

1.2.3 Sección de código de usuario

En esta sección se debe colocar todo el código en `C` que el usuario desee colocar como pueden ser: una función `main`, o definición de subrutinas.

2 Bison

Bison es un generador de analizadores sintácticos de propósito general que convierte una descripción gramatical para una gramática independiente del contexto `LALR` en un programa en `C` que analice esa gramática.

2.1 Instalación

Para sistemas basados en Debian ingrese el siguiente comando en su terminal:

```
usuario@pc:~$ sudo aptitude install bison
```

Verifique que la instalación se realizó con éxito, comprobando la versión de bison

```
usuario@pc:~$ bison --version
```

2.2 Etapas en el Uso de Bison

El proceso real de diseño de lenguajes utilizando Bison, desde la especificación de la gramática hasta llegar a un compilador o intérprete funcional, se compone de estas etapas:

1. Especificar formalmente la gramática en un formato que reconozca Bison. Para cada regla gramatical en el lenguaje, describir la acción que se va a tomar cuando una instancia de esa regla sea reconocida. La acción se describe por una secuencia de sentencias en `C`.

2. Escribir un analizador léxico para procesar la entrada y pasar tokens al analizador sintáctico. El analizador léxico podría escribirse a mano en `C` o puede generarse utilizando `Flex`

3. Escibir una función de control que llame al analizador producido por Bison.

4. Escribir las rutinas de infome de errores.

Para hacer que este código fuente escrito se convierta en un programa ejecutable, debe seguir estos pasos:

1. Ejecutar Bison sobre la gramática para producir el analizador.
2. Compilar el código de salida de Bison, al igual que cualquier otro fichero fuente.
3. Enlazar los ficheros objeto para producir el producto final.

2.3 Formato de una Gramática de Bison

El fichero de entrada para la utilidad Bison es un archivo de gramática de Bison. La forma general de una gramática de Bison es la siguiente:

```
%{
declaraciones en C
%}
Declaraciones de Bison
%%
Reglas gramaticales
%%
Código C adicional
```

Las declaraciones en `C` podrían definir tipos y variables utilizadas en las acciones. Puede también usar comandos del preprocesador para definir macros que se utilicen ahí, y utilizar `#include` para incluir archivos de cabecera que realicen cualquiera de estas cosas.

Las declaraciones de Bison declaran los nombres de los símbolos terminales y no terminales, y también podrían describir la precedencia de operadores y los tipos de datos de los valores semánticos de varios símbolos.

Las reglas gramaticales definen cómo construir cada símbolo no terminal a partir de sus partes.

El código `C` adicional puede contener cualquier código que desee utilizar. A menudo suele ir la definición del analizador léxico yylex, más subrutinas invocadas por las acciones en la reglas gramaticales. En un programa simple, todo el resto del programa puede ir aquí.

3 Caso de Estudio

Se plantea realizar un analizador léxico y sintáctico que reciba como entrada un archivo con código fuente y determine, según ciertas reglas, si este posee errores sintácticos. La salida generada en caso de error, será dado por el patrón:

```
Error en la línea [Nro.  de línea]: [Indicación del error]
```

3.1 Estructura del caso de estudio

La codificación del caso de estudio se estructuro en los siguientes archivos, los cuales se procederán a explicar según su grado de importancia para el resultado:

```
AnalizadorSintáctico
├── analizador.l
├── analizador.y
├── ejemplo.c
├── funcion_prueba.c
└── Makefile
```

3.1.1 analizador.l

Es un archivo de `Flex`, divido en tres secciones separadas por '% %'. En la parte superior del código se incluyen las bibliotecas necesarias para el `main`. Seguidamente se define la segunda sección que contienes las expresiones regulares que servirán para identificar elementos del lenguaje `C`, por ejemplo `letra [a-zA-Z_]` que indica el emparejamiento cualquier letra minúscula o mayúscula de la **A** a la **Z**. Y por último la sección de las reglas que seguirá el analizador léxico, dónde cada `Patrón` puede usar las definiciones de las restricciones de las `Acciones` (fragmentos de código de expresiones básicas, caracteres o símbolos de `C`).

```
%{
#include <stdio.h>
#include <stdlib.h>
#include "analizador.tab.h"
int lineas=0;
%}

/* Definimos las expresiones regulares que serviran
 * para identificar elementos del lenguaje C */

letra [a-zA-Z_]
DIGITO [0-9]
ID ([a-zA-Z]|"_")([a-zA-Z0-9]|"_")*
TIPOSDATO "char"|"int"|"long"|"short"|"float"|"double"|"void"
OPRELACION "<"|">"|">="|"<="|"!="|"=="
OPINCREMENTO "+"|"-"|"*"|"/"
OPINCREMENTO2 "++"|"--"
OPINCREMENTO3 "-="|"+="|"*="|"/="
OPLOG "&&"|"||"
%%

"else" {yylval.valor = strdup(yytext); return(T_ELSE);}
"switch" {yylval.valor = strdup(yytext); return(T_SWITCH);}
"case" {yylval.valor = strdup(yytext); return(T_CASE);}
"for" {yylval.valor = strdup(yytext); return(T_FOR);}
"while" {yylval.valor = strdup(yytext); return(T_WHILE);}
"do" {yylval.valor = strdup(yytext); return(T_DO);}
"if" {yylval.valor = strdup(yytext); return(T_IF);}
"return" {yylval.valor = strdup(yytext); return(T_RETURN);}
```

```
"break" {yylval.valor = strdup(yytext); return(T_BREAK);}
"struct" {yylval.valor = strdup(yytext); return(T_STRUCT);}
{DIGITO}+{yylval.valor = strdup(yytext); return(T_INT);}
{DIGITO}+"."{DIGITO}+{yylval.valor = strdup(yytext); return(T_FLOAT);}
"'"{letra}"'"{yylval.valor = strdup(yytext); return(T_CAR);}
"\""{letra}"\""{yylval.valor = strdup(yytext); return(T_CHAR);}
{TIPOSDATO} {yylval.valor = strdup(yytext); return(T_TDATO);}
{ID} {yylval.valor = strdup(yytext); return(T_ID);}
{OPINCREMENTO} {yylval.valor = strdup(yytext); return(T_INCR);}
{OPINCREMENTO3} {yylval.valor = strdup(yytext); return(T_OPAARIT);}
{OPINCREMENTO2} {yylval.valor = strdup(yytext); return(T_INCR2);}
{OPRELACION} {yylval.valor = strdup(yytext); return(T_OPRELACION);}
{OPLOG} {yylval.valor = strdup(yytext); return(T_ANDOR);}
";" {yylval.valor = strdup(yytext); return(T_PTCOM);}
"," {yylval.valor = strdup(yytext); return(T_COMA);}
":" {yylval.valor = strdup(yytext); return(T_DOSPTO);}
"=" {yylval.valor = strdup(yytext); return(T_OPASIG);}
"(" {yylval.valor = strdup(yytext); return(T_PA);}
")" {yylval.valor = strdup(yytext); return(T_PC);}
"{" {yylval.valor = strdup(yytext); return(T_CA);}
"}" {yylval.valor = strdup(yytext); return(T_CC);}
\n      ++lineas;
.   {}
%%
```

3.1.2 analizador.y

Es un archivo de fuente `Bison`, el cual se divide en tres secciones principales que igualmente están separadas por ' % %'. (código analizador.y)

1. Declaraciones: Se declaran todas las variables y librerias a utilizar en el programa, declaraciones necesarias para `Bison` y para C, se declaran los terminales y no terminales de la gramatica, ademas de los tokken que se reciben desde el analizador léxico.

```
%{
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
extern int yylex(void);
extern char *yytext;
extern int lineas;
extern FILE * yyin;
FILE * fsalidaRep;
FILE * fsalidaSel;
```

```
FILE * fsalidaFun;
FILE * fsalidaStru;
void yyerror(char *s);
%}

%union
{
char *valor;
}

%token <valor> T_ID T_INT T_FLOAT T_IF T_ELSE T_SWITCH T_CASE T_FOR \\
 T_WHILE T_DO T_TDATO T_OPAARIT T_INCR2 T_COMA T_PTCOM T_OPRELACION \\
 T_OPASIG T_PA T_PC T_CA T_CC T_INCR T_ANDOR T_CHAR T_RETURN \\
 T_STRUCT T_DOSPTO T_BREAK T_CAR
%type <valor> argumentos
%type <valor> declaracion
%type <valor> id
%type <valor> conte
%type <valor> return
%type <valor> asignaciones
%type <valor> operArit
%type <valor> valores
%type <valor> cases
%type <valor> case
%type <valor> constantes
%start funciones
%%
```

2. Reglas Gramaticales: Se declaran las definiciones de las diferentes reglas gramaticales que componen la sintaxis del programa, definiendo un orden para los tokken (terminales o no terminales) dependiendo de cada una de las reglas. Estas son de la forma:

(regla): (producción) | (producción) ...

```
valores: T_INT {$$=$1;}|
         T_FLOAT {$$=$1;}|
         T_ID {$$=$1;};
```

Para este caso la regla contiene no terminales; números de tipo entero, flotante o variables declaradas, dentro del bloque de {} se ejecutan acciones para validar o asignar los valores correspondientes del tokken; a través del operador '$$' y '$n' donde n es el numero de posición del elemento al cual se quiere acceder en la producción.

Cada definición como ya vimos puede tener mas de una producción asociada, separando cara una con el carácter "|" pero ademas de esto como producciones se pueden definir recuperación a errores

de sintaxis comunes, las cuales simplemente se agregan como producciones “erradas” a la definición, como por ejemplo:

```
declaracion: T_TDATO id T_PTCOM {strcat($1," ");strcat($1,$2);$$=$1;}|
             T_TDATO id T_PTCOM declaracion     {strcat($1," "); \\
             strcat($1,$2);strcat($1,", ");strcat($1,$4);$$=$1;}|
/*Excepciones declaraciones*/
         T_TDATO id {printf (" Error en linea %d: Te falto poner el ;
en la declaracion. \n", lineas-1);}
```

3. Código de usuario: en esta ultima sección, se escribe todo el código adicional o funciones que requiera el analizador, en este caso se valida dentro de la función `main` que los parámetros de al momento de compilar estén completos, y se abren los archivos donde se guarda la información de la compilación.

```
int main( int argc, char **argv )
{
        if (argc>1){
           yyin = fopen(argv[1], "r");
           fsalidaRep = fopen("Estructura Repetitiva.txt","w");
           fsalidaSel = fopen("Estructura Selectiva.txt","w");
           fsalidaFun = fopen("Funciones.txt","w");
           fsalidaStru = fopen("Structs.txt","w");
        }else{
          printf("Forma de uso: ./analizador archivo_entrada\n");
          return 0;
        }
        yyparse();

        return 0;
}
```

3.1.3 Ejemplo del Analizador sintáctico

Ahora les vamos a enseñar como compilar nuestro analizador sintactico y les vamos a mostrar varios ejemplos para que vean su funcionamiento.
Primero vamos a ejecutar el comando make, Al ejecutar este comando automaticamente se van a generar nuevos archvios con flex se genera lex.yy.c, este es el primer archivo luego se van a generear las librerias que serian analizador.tab.c y analizador.tab.h esto es con bison y tambien se va a generar nuestro ejecutable el cual se llama analizador, para usar nuestro analizador tenemos que escribir en la consola `./analizador y nuestro arvhivo de entrada.`
el primer archivo de entrada que se va a cargar es ejemplo.c

```c
int main(){

        int x, y;

        int p;

        x= 5+7;
        x= 0;
        y = 9;


        struct n
        {
                int a;
                float b;
                char c;
        };


        switch(v)
        {
                case 5: x = 0; v++; break;
                case 'a': x = 0; v++; break;
        }


        if (x>0)
        {
                for (y=1; y<6;y++)
                {
                        x++;
                }
        }

        else
        {
                y=6;
                while(y>0){
                x++;
                y--;
                }
        }

        for (y=1; y<6;y++)
        {
```

```
46                  for (y=1; y<6;y++)
47                  {
48                          x++;
49                  }
50          }
51
52
53          if (x>0)
54          {
55                  for (y=1; y<6;y++)
56                  {
57                          x++;
58                  }
59          }
60
61  return (0);
62  }
63
64  int funcion(int i, float f)
65  {
66          i=0;
67          f=i+1;
68
69  struct nuevaEstructura
70  {
71                  int aX;
72                  float flotante;
73                  char ccar;
74  };
75
76  return (9);
77  }
78
79  void nulo()
80  {
81          do
82          {
83                  x = x-1;
84          }while(x > 0);
85  }
```

ejemplo.c compilo perfectamente esto quiere decir que no tiene ningun error sintactico, como pueden observar se generaron nuevos archivos .txt:structura repetitiva, estructura selectiva, funciones y strings. Esos archivos.txt contienen una breve descripcion del archivo.c
posteriormente se agregaron varios errores al codigo para mostrar como funciona el analizador sintactico, asi poder demostrar como funciona la recuperacion de errores con las producciones que se

colocaron en el analizador.h

```
int main()
{

        int x, y;

        int p

        x= 5+7;
        x= 0;
        = 9;


        struct n
        {
                int a;
                float b;
                char c;
        };


        switch(v)
        {
                case 5: x = 0; v++; break;
                case 'a': x = 0; v++; break;
        }


        if (x>0)
        {
                for (y=1; y<6;y++)
                {
                        x++;
                }


        else
        {
                y=6;
                while(y>0)
                {
                        x++;
                        y--;
                }
        }
```

```
45
46          for (y=1; y<6;y++)
47          {
48                  for (y=1; y<6;y++)
49                  {
50                          x++;
51                  }
52          }
53
54
55          if (x>0)
56          {
57                  for (y=1; y<6;y++)
58                  {
59                          x++;
60                  }
61          }
62
63  return (0);
64  }
65
66  int funcion(int i, float f)
67  {
68          i=0;
69          f=i+1;
70
71  struct nuevaEstructura
72  {
73          int aX;
74          float flotante;
75          char ccar;
76  };
77
78  return (9);
79  }
80
81  void nulo()
82  {
83          do
84          {
85                  x = x-1;
86          }while(x > 0);
87  }
```

Al compilar y ejectuar el codigo apareceran errores en las lineas 5,9 y 51 lo cual indica que nuestra recuperacion de errores esta funcionando porque no se esta cortando la compilacion del programa y

esta mostrando los errores, despues de corregir los errores volvemos a compilar y efectivamente compila sin problemas.

Ahora vamos a usar otro ejemplo, en este caso sera `funcion_prueba.c`

```
1         int main()
2         {
3                 int x, y, z, cont
4
5                 x = 0;
6                 = 0;
7
8
9                 for (cont = 1; cont <= 20; cont = cont+1)
10
11                        z = x+y;
12                        x = y;
13                        y = z;
14                }
15        return (0);
16}
```

Esta funcion ya tiene varios errores, pasamos a compilar para que los reconozca, y efectivamente el analizador muestra los errores en la linea 3, 6 y 10.
luego de corregir estos errores.

```
1 int main(){
2
3         int x, y, z, cont;
4
5         x = 0;
6         y = 0;
7
8
9         for (cont = 1; cont <= 20; cont = cont+1)
10        {
11
12                z = x+y;
13                x = y;
14                y = z;
15        }
16return (0);
17}
```

se vuelve a compilar y el archivo funciona sin problemas. Esto fue todo nuestro analizador sintactico,

esperamos que les haya gustado

Referencias

[1] Campues (2019). Investigacion Flex Y Bison. Es.slideshare.net. Tomado de: https://es.slideshare.net/jvalexander/investigacion-flex-y-bison.

[2] Es.tldp.org. (2019). Bison 1.27. Tomado de: http://es.tldp.org/Manuales-LuCAS/BISON/bison-es-1.27.html.

[3] CCM. (2019). How To Install Flex and Bison Under Ubuntu. Tomado de: https://ccm.net/faq/30635-how-to-install-flex-and-bison-under-ubuntu.