Hoja técnica sobre `yacc`[1] y `lex`[2]

Introducción

Los programas `yacc` y `lex` son herramientas de gran utilidad para un diseñador de compiladores. Muchos compiladores se han construido utilizando estas herramientas (p.ej., el compilador de C de GNU (`gcc`)) o versiones más avanzadas. Los programas `bison` y `flex` son las versiones más modernas (no comerciales) de `yacc` y `lex`, y se distribuyen bajo licencia GPL con cualquier distribución de Linux (y también están disponibles para muchos otros UNIX).

`lex`

El programa `lex` genera analizadores léxicos a partir de una especificación de los componentes léxicos en términos de expresiones regulares (en el estilo de UNIX); `lex` toma como entrada un fichero (con la extensión `.l`) y produce un fichero en C (llamado "`lex.yy.c`") que contiene el analizador léxico. El fichero de entrada tiene tres partes (ver los ejemplos al final de este documento):

```
definiciones
%%
expresiones regulares - acciones
%%
código
```

1. Al principio del fichero puede aparecer una parte de definiciones auxiliares.

   a) Una definición ocupa una línea y está formada por un identificador, uno o más espacios en blanco o tabuladores y una expresión regular. La sintaxis de las expresiones regulares se define en la tabla 3.48 del libro de Aho, Sethi y Ullman (en el ejercicio 3.10). Se puede hacer referencia más adelante a una definición regular poniendo el identificador encerrado entre llaves[3].

   b) En la parte de definiciones auxiliares pueden aparecer fragmentos de código (que debe ser C o C++; `lex` no lo comprueba) encerrados entre "`%{`" y "`%}`". En esos fragmentos de código se pueden poner declaraciones de variables globales y funciones que se utilicen más adelante. Los fragmentos de código pueden aparecer en cualquier línea de esta primera parte, antes y/o después de las definiciones regulares.

2. Después de la parte de definiciones auxiliares (que puede no tener nada) debe aparecer el símbolo "`%%`" y una secuencia de cero o más pares expresión regular – acción. Estas expresiones regulares deben representar los componentes léxicos del lenguaje que se quiere reconocer. Cada acción es un fragmento en C que debe estar encerrado entre llaves[4].

   Normalmente, las acciones consistirán en incrementar los contadores de líneas y columnas, y hacer un `return` del *token* correspondiente. Si no se hace `return`, el analizador léxico sigue buscando otro componente léxico.

3. Después de todas las expresiones y sus acciones puede aparecer más código en C. En este caso debe aparecer el símbolo "`%%`" antes del código (que no debe estar encerrado entre "`%{`" y "`%}`").

Notas sobre el comportamiento de `lex`:

- En el caso de que una cadena de texto pueda ser reconocida por dos o más expresiones regulares (como p.ej. las palabras reservadas, que normalmente también son reconocidas por la expresión regular que reconoce identificadores), `lex` tomará la que aparezca antes en el fichero.

[1] Johnson, S.C. (1975) "Yacc: Yet Another Compiler Compiler", Computing Science Technical Report No. 32, AT&T Bell Labs., Murray Hill, New Jersey.

[2] Lesk, M.E., Schmidt, E. (1975) "Lex - A Lexical Analyzer Generator", Computing Science Technical Report No. 39, AT&T Bell Labs., Murray Hill, New Jersey.

[3] En algunas versiones de `lex` solamente se puede hacer referencia a definiciones regulares en la segunda parte del fichero, pero como parte de otra definición. Afortunadamente, `flex` sí permite utilizar las definiciones para construir otras definiciones regulares.

[4] `lex` elimina las llaves y construye una función con un `switch` que ejecuta cada acción. Si es necesario declarar variables locales a esa función se puede hacer poniendo un fragmento de código encerrado entre "`%{`" y "`%}`" *después* del símbolo "`%%`", antes de cualquier par expresión regular – acción.

- El analizador léxico que genera `lex` reconoce siempre la cadena *más larga* que coincida con una de las expresiones regulares, aunque un prefijo de esa cadena sea reconocido por otras expresiones regulares.
- Si un carácter no puede ser reconocido por ninguna expresión regular, `lex` lo imprime por la salida estándar. Esta función está pensada para que `lex` pueda actuar como *filtro* con ficheros de texto[5], pero para implementar un compilador esta función debe ser controlada, y para eso la última expresión suele ser un punto (que reconoce cualquier caracter excepto `\n`) y la acción correspondiente suele producir un mensaje de error.
- Las expresiones regulares deben aparecer al principio de la línea para ser reconocidas por `lex`; si aparecen más a la derecha `lex` pensará que son parte de las acciones.
- En general, `lex` no comprueba que el código sea C, C++ o cualquier otro lenguaje; simplemente lo inserta en el lugar que le corresponde en el analizador léxico que va a generar. Este analizador léxico se construye a partir de un *esqueleto* que suele estar escrito en C.

El fichero en C que genera `lex` contiene una función `yylex` que realiza la función del analizador léxico, y devuelve un `int` que debe ser el número que representa cada *token* del lenguaje; cuando llega al final de la entrada devuelve cero.

El fichero `lex.yy.c` se puede compilar (si se añaden los `#define` correspondientes a los *tokens* que devuelve), pero el *linker* dará error porque necesita dos funciones: `main` y `yywrap`. Para solucionar este problema se puede indicar al *linker* que utilice la librería del `lex` (con la opción `-ll` al final) o bien se pueden definir las funciones en la parte de código al final de la especificación de expresiones regulares; si se elige esta opción, se debe tener en cuenta que la función `yywrap` no tiene argumentos y debe devolver un entero, que normalmente debe ser un 1[6].

Por defecto, el analizador léxico generado por `lex` lee caracteres de la entrada estándar `stdin`. Si se desea cambiar este comportamiento basta con hacer que la variable `yyin` (que es de tipo `FILE *`) apunte a otro fichero (previamente abierto para lectura con `fopen`) antes de llamar a `yylex`.

Condiciones de arranque en `lex`

En determinadas circunstancias, puede ser conveniente que el analizador léxico entre en un estado específico para reconocer partes del lenguaje que dependen de un contexto. El ejemplo más común es el de los comentarios de varias líneas en C y/o Pascal. Cuando se reconoce el comienzo del comentario, el analizador entra en un estado especial (determinado por una *condición de arranque*) en el que reconoce el comentario de varias líneas y en el que no se reconocen otras expresiones regulares del analizador. Cuando el comentario termina, el analizador vuelve al estado original `INITIAL` y sigue reconociendo tokens.

```
/* condición de arranque para el comentario */
%x COMENTARIO

%%

"+"                     { return ADDOP; }
"/*"                    {BEGIN(COMENTARIO);}
<COMENTARIO>[^/*\n]+    {/* cualquier secuencia de caracteres que no tenga * / ni \n */}
<COMENTARIO>\n          {/* hay que seguir contando líneas */}
<COMENTARIO><<EOF>>     {/* error, fin de fichero encontrado en el comentario */}
<COMENTARIO>"*/"        {/* volvemos al estado inicial */  BEGIN(INITIAL);}
<COMENTARIO>[*/]        {/* * y / sueltos (p.ej. ****/)}
"print"                 {return PRINT;}
```

yacc

A partir de un fichero de especificación (bastante parecido a un ETDS), `yacc` genera un fichero en C (llamado "`y.tab.c`"), que contiene un analizador sintáctico ascendente por desplazamiento y reducción[7], y una función que

[5] Por ejemplo, se puede utilizar para convertir todas las palabras en mayúsculas a palabras en minúsculas, ignorando lo que no sean palabras.

[6] La función `yywrap` es llamada por el analizador léxico cuando alcanza el final del fichero; si devuelve 1, termina el análisis y si devuelve 0 continua. Esta función se puede utilizar para procesar varios ficheros en una pasada (*includes*). En la documentación del `lex` se explica mejor el funcionamiento de `yywrap`.

[7] Las tablas de análisis las construye según el método LALR, más complejo que el SLR, pero que genera tablas más compactas que el LR canónico (ver la sección 4.7 del libro de Aho, Sethi y Ullman).

ejecuta las acciones semánticas que aparecen en el fichero de especificación. Este fichero (que normalmente llevará la extensión `.y`) tiene tres partes, separadas igual que en el fichero para `lex` con el símbolo "`%%`":

```
definiciones de tokens, código auxiliar
%%
reglas - acciones
%%
código
```

1. La primera parte es una secuencia de definición de *tokens*. Cada definición ocupa una línea, y empieza con la cadena "`%token`" y una secuencia de uno o más identificadores (los nombres de los tokens o terminales de la gramática). Por cada *token* definido `yacc` generará una línea parecida a "`#define NINT 260`", numerando los *tokens* a partir del 256. Existen otras formas de definir *tokens* en las que además se puede especificar su asociatividad y precedencia (para resolver conflictos provocados por la utilización de gramáticas ambiguas), pero no es necesario utilizarlas en las prácticas de la asignatura.

   En esta primera parte es posible incluir fragmentos de código encerrados entre "`%{`" y "`%}`", como en los ficheros de especificación para `lex`. Todo este código es global a todo el fichero.

2. La segunda parte del fichero debe contener las reglas de la gramática y las acciones semánticas, en una notación muy similar a los ETDS. Los no terminales se representan con identificadores, los terminales se ponen tal como se hayan definido en la primera parte y las acciones se deben encerrar entre llaves. Igual que `lex`, `yacc` elimina las llaves y pone todo el código de las acciones en una única función dentro de un `switch`. A diferencia de `lex`, no es posible declarar variables locales a esa función (al menos no es posible con `bison`).

   Cada regla de la gramática debe empezar por un no terminal al principio de la línea, seguido del símbolo '`:`' (dos puntos) y la parte derecha de la regla con terminales, no terminales y acciones separados por espacios en blanco, tabuladores o `\n`. Si se desea definir más reglas de la misma variable se puede poner el símbolo "`|`" (barra vertical) y otra parte derecha. Después de la última parte derecha debe aparecer el símbolo "`;`" (punto y coma). Las partes derechas de las reglas pueden partirse en varias líneas si es necesario, y normalmente no continuan en la primera columna.

3. La tercera parte contiene código en C o C++ (que no debe estar encerrado entre "`%{`" y "`%}`"). En este código *deben* definirse dos funciones al menos: `main` y `yyerror`. El argumento de `yyerror` es una cadena de caracteres (`char *`); cuando el analizador generado por `yacc` encuentra un error sintáctico llama a la función `yyerror` con la cadena "`parse error`"[8].

El fichero generado por `yacc` contiene una función `yyparse` que llama a `yylex` (si no se utiliza `lex` se debe definir una función con ese nombre que devuelva un `int`, que realice las funciones del analizador léxico) cada vez que necesita un *token*), analiza sintácticamente el fichero fuente y ejecuta las acciones en el momento adecuado. Es necesario tener en cuenta que `yacc` genera sus propios *marcadores* para las acciones que aparezcan antes del final de la parte derecha de la regla[9], por lo que no es necesario introducir explícitamente esos marcadores en el ETDS antes de proporcionárselo al `yacc`; además, se puede aprovechar el espacio ocupado por dichos marcadores en la pila de atributos para almacenar atributos heredados o valores locales a una regla[10].

Los valores que debe devolver el analizador léxico son los que genera el `yacc` a partir de la primera parte de definición de *tokens*.

Atributos con `yacc`

La función `yyparse` gestiona una pila paralela a la del análisis sintáctico. Esa pila contiene elementos del tipo `YYSTYPE`, que por defecto está definido a `int`. Para acceder a esta pila en las acciones se debe utilizar una notación especial (que `yacc` traduce a código en C):

- Los símbolos de la parte derecha se numeran del uno en adelante (las acciones en medio de las partes derechas ocupan el espacio de un símbolo y es necesario tenerlo en cuenta).

[8] El programa `bison` puede generar una cadena que proporciona más información sobre la causa del error, pero para realizar las prácticas se debe ignorar esa cadena y elaborar el mensaje de error como se especifica en el enunciado de la práctica.

[9] Puesto que `yacc` sustituye cada acción por un no terminal marcador (Aho, Sethi y Ullman 1990, sección 5.6), hay que tener en cuenta que ese marcador está implícito en la parte derecha de la regla y por tanto hay un símbolo más por cada acción que aparezca en la parte derecha.

[10] Si en una acción intermedia se le asigna un valor a `$$`, realmente se está utilizando los atributos del marcador, no los de la parte izquierda de la regla (aunque en una acción al final se utilice `$$` para hacer referencia a la parte izquierda)

- Para hacer referencia a la posición en la pila de un símbolo se pone (dentro del código en C) `$n`, donde `n` es el número de ese símbolo en la parte derecha.
- Para hacer referencia a la parte izquierda se pone `$$` (sólo en acciones al final de la parte derecha, justo antes de que el analizador reduzca por la regla[11]).
- Es muy aconsejable situar las acciones semánticas al final de las reglas. Cuando no sea posible, es recomendable situarlas siempre entre terminales o variables que no generen ϵ; si se colocan después de una variable que puede derivar a ϵ es probable que el `yacc` no pueda decidir cuándo realizar la acción y produzca conflictos reducción-reducción.

Para hacer que `YYSTYPE` sea un tipo distinto de `int` es suficiente con poner `#define YYSTYPE MITIPO` en un fragmento de código en la primera parte del fichero (junto con las definiciones de los *tokens*), donde `MITIPO` es normalmente un `struct` con los atributos.

Atributos heredados

Al utilizar un analizador sintáctico ascendente no es posible implementar directamente un ETDS con atributos heredados en `yacc`; es necesario conseguir que cada atributo heredado se corresponda con un atributo sintetizado de otra variable que esté más abajo en la pila (y para hacer estas transformaciones —ver el epígrafe 5.6 del libro de Aho, Sethi y Ullman— sí puede ser necesario utilizar marcadores). Para acceder a atributos que se encuentren debajo del primer símbolo de la parte derecha habría que utilizar los índices `$0`, `$-1`, etc., siempre teniendo en cuenta que las acciones ocupan un hueco en la pila.

Variables locales a una regla

Cuando se desea utilizar variables locales a una regla, es decir, variables locales al proceso de análisis de una regla, no es posible utilizar variables locales de C, ya que son globales a todas las reglas (y a todos los procesos de análisis de cada regla), ya que es una única función de C la que ejecuta todas las acciones. En su lugar se deben utilizar atributos de algún símbolo de la parte derecha de la regla (que ya se haya analizado, por supuesto); los terminales son los candidatos más adecuados para esa tarea, aunque también se pueden utilizar marcadores o los marcadores de las propias acciones. Si se utiliza esta técnica se debe documentar con comentarios en el código.

Comunicación entre `yacc` y `lex`

Los programas que son generados por `yacc` y `lex` están diseñados para funcionar juntos. Así, `lex` debe devolver los números de *token* que asigna `yacc`. Para ello es necesario ejecutar `yacc` con la opción `-d`, que hace que `yacc` genere un fichero `y.tab.h` que contenga sólo los *defines* de los tokens. Ese fichero debe ser incluído con `#include` en el fichero de `lex`, al principio, para que los tokens sean conocidos dentro del código generado por `lex`.

Además de devolver un entero indicando qué *token* se ha reconocido, el analizador léxico debe devolver el valor léxico de ese *token*. El lexema que el analizador generado por `lex` ha reconocido se copia en la cadena de caracteres `yytext` (y no `yytexto` como pone en la traducción a castellano del libro de Aho, Sethi y Ullman, en la página 113). El analizador léxico debe procesar esa cadena de caracteres (convertirla a número, etc.) y devolver al analizador sintáctico el valor léxico del *token* (que no tiene por qué coincidir con el lexema). Para hacer esto con `yacc` y `lex` existe una variable `YYSTYPE yylval`[12] que debe rellenar el analizador léxico[13] y que `yacc` copia en la pila semántica cuando desplaza el símbolo a la pila sintáctica.

Ejemplo con `yacc` y `lex`

El siguiente ejemplo es una implementación de un traductor sencillo utilizando `yacc` y `lex`. Para generar el ejecutable hay que teclear:

[11] Si se utiliza `$$` en alguna acción antes del final de la regla, en realidad se estará utilizando el espacio ocupado por la acción (sería lo mismo que utilizar `$n`, siendo `n` la posición de la acción en la regla), lo cual puede (a veces) ser conveniente.

[12] Dependiendo de la versión de `yacc` y `lex` está declarada en el código que genera `yacc` o en el código que genera `lex`.

[13] Las acciones que se ejecutan cuando se reconoce un componente léxico deben asignar un valor adecuado a esa variable antes de devolver con `return` el número correspondiente a ese componente léxico.

```
flex ejemplo.l
bison -d ejemplo.y
g++ -o ejemplo ejemplo.tab.c lex.yy.c
```

```
/*---------------------- comun.h -----------------------------*/

/* fichero con definciones comunes para los ficheros .l y .y */

typedef struct {
   char *lexema;
   int nlin,ncol;
   int tipo;
   string cod;
} MITIPO;

#define YYSTYPE MITIPO


#define ERRLEXICO    1
#define ERRSINT      2
#define ERREOF       3
#define ERRLEXEOF    4

void msgError(int nerror,int nlin,int ncol,const char *s);


/*------------------------------ ejemplo.l --------------------------------*/
D    [0-9]
L    [a-zA-Z]
LD   [0-9a-zA-Z]

%{
#include <string.h>
#include <string>
#include <iostream>

using namespace std;

#include "comun.h"
#include "ejemplo.tab.h"

int ncol = 1,
    nlin = 1;

int findefichero = 0;

int ret(int token);
// función que actualiza 'nlin' y 'ncol' y devuelve el token

void msgError(int nerror, int nlin,int ncol,const char *s);
// función para producir mensajes de error

%}

%x COMENTARIO

%%
%{
/* codigo local */
%}

" "                {ncol++;}
[\t]               {ncol++;}
[\n]               {nlin++;ncol=1;}
```

```
"//"(.)*           {; /* comentarios de una linea, no hacer nada */}
"/*"                    {ncol += strlen(yytext);BEGIN(COMENTARIO);}
<COMENTARIO>[^/*\n]+    {ncol += strlen(yytext);}
<COMENTARIO>\n          {nlin++;ncol=1;}
<COMENTARIO><<EOF>>     {msgError(ERRLEXEOF,-1,-1,"");}
<COMENTARIO>"*/"        {ncol+=strlen(yytext); BEGIN(INITIAL);}
<COMENTARIO>[*/]        {ncol += strlen(yytext);}
"print"            {return ret(print);}
        /* Las palabras reservadas deben aparecer antes de la regla que
          reconoce los identificadores, para evitar que sean reconocidas
          como identificadores en lugar de como palabras reservadas */
{L}({LD})*         {return ret(id);}
{D}+               {return ret(numentero);}
{D}+(\.){D}+       {return ret(numreal);}
"+"                {return ret(opas);}
"-"                {return ret(opas);}
"("                {return ret(pari);}
")"                {return ret(pard);}
";"                {return ret(pyc);}
","                {return ret(coma);}
.                  {msgError(ERRLEXICO,nlin,ncol,yytext);}

%%

int yywrap(void) {findefichero=1; return 1;} /* para no tener que linkar con la
                                libreria del lex */


int ret(int token)
{
   yylval.lexema=strdup(yytext);
   yylval.nlin=nlin;
   yylval.ncol=ncol;
   ncol+=(strlen(yytext));
   return(token);
}
```

```
/*------------------------------ ejemplo.y -------------------------------*/
%token print id
%token opas opmd
%token numentero numreal pari pard
%token pyc coma

%{

#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#include <string>
#include <iostream>


using namespace std;

#include "comun.h"

// variables y funciones del A. Léxico
extern int ncol,nlin,findefichero;


extern int yylex();
extern char *yytext;
extern FILE *yyin;
```

```
int yyerror(char *s);


const int ENTERO=1;
const int REAL=2;

string operador, s1, s2;  // string auxiliares

%}

%%


S    : print SExp pyc    { /* comprobar que después del programa no hay ningún token más */
                           int tk = yylex();
                           if (tk != 0) yyerror("");
 }
     ;


SExp : SExp coma Exp     { cout << $3.cod << endl; }
     | Exp               { cout << $1.cod << endl; }
     ;


Exp : Exp opas Factor    { if (!strcmp($2.lexema,"+"))
                                operador = "sum";
                           else
                                operador = "res";
                           if ($1.tipo != $3.tipo)
                           {
                                 if ($1.tipo == ENTERO)
                                    s1 = "itor(" + $1.cod + ")";
                                 else
                                    s1 = $1.cod;
                                 if ($3.tipo == ENTERO)
                                    s2 = "itor(" + $3.cod + ")";
                                 else
                                    s2 = $3.cod;
                                 operador +="r";
                                 $$.tipo = REAL;
                                 $$.cod = operador + "(" + s1 + "," + s2 + ")";
                           }
                           else
                           {
                                 s1 = $1.cod;
                                 s2 = $3.cod;
                                 if ($1.tipo == REAL)
                                    operador += "r";
                                 else
                                    operador += "i";
                                 $$.tipo = $1.tipo;
                                 $$.cod = operador + "(" + s1 + "," + s2 + ")";
                           }
                         }
        | Factor         /* $$ = $1 */
        ;

Factor : numentero           { $$.tipo = ENTERO;
                               $$.cod = $1.lexema;
```

```
                            }
       | numreal            { $$.tipo = REAL;
                              $$.cod = $1.lexema;
                            }
       | pari Exp pard      { $$.tipo = $2.tipo;
                              $$.cod = $2.cod;
                            }
       | id                 { $$.tipo  = ENTERO; // todas las variables son enteras
                              $$.cod = $1.lexema;
                            }
       ;

%%

void msgError(int nerror,int nlin,int ncol,const char *s)
{
     switch (nerror) {
         case ERRLEXICO: fprintf(stderr,"Error lexico (%d,%d): caracter '%s' incorrecto\n",nlin,ncol,s);
            break;
         case ERRSINT: fprintf(stderr,"Error sintactico (%d,%d): en '%s'\n",nlin,ncol,s);
            break;
         case ERREOF: fprintf(stderr,"Error sintactico: fin de fichero inesperado\n");
            break;
         case ERRLEXEOF: fprintf(stderr,"Error lexico: fin de fichero inesperado\n");
            break;
     }

     exit(1);
}


int yyerror(char *s)
{
    if (findefichero)
    {
       msgError(ERREOF,0,0,"");
    }
    else
    {
       msgError(ERRSINT,nlin,ncol-strlen(yytext),yytext);
    }
}

int main(int argc,char *argv[])
{
    FILE *fent;

    if (argc==2)
    {
        fent = fopen(argv[1],"rt");
        if (fent)
        {
            yyin = fent;
            yyparse();
            fclose(fent);
        }
        else
            fprintf(stderr,"No puedo abrir el fichero\n");
    }
    else
        fprintf(stderr,"Uso: ejemplo <nombre de fichero>\n");
}
```