

## Analizador Sintáctico para el lenguaje BASIC.

### Listings

1	Analizador Sintáctico . . . . .	6
2	Analizador Léxico . . . . .	8
3	Archivo de Pueba 1 . . . . .	12
4	Archivo de Prueba 2 . . . . .	12

### List of Figures

1	Construir un compilador/traductor usando Lex & Yacc (Niemann) . . . . .	2
2	Comandos para crear el compilador (Niemann) . . . . .	2
4	Terminal de Linux en el directorio de archivos fuente . . . . .	10
6	Salida en consola al ejecutar el Analizador Sintáctico . . . . .	13

### List of Tables

1	Tabla de Gramatica para el A. Sintáctico . . . . .	15
---	--	----

## 1 Analizador Sintáctico

### 1.1 Introducción

Todo lenguaje de programación obedece a unas reglas que describen la estructura sintáctica de los programas bien formados que acepta. Se puede describir la sintaxis de las construcciones de los lenguajes de programación por medio de gramáticas de contexto libre o utilizando notación BNF (*Backus-Naur Form*).

## 1.2 Lex & Yacc - Introducción

Para construir un compilador/traductor con *Lex* & *Yacc* debemos de analizar el diagrama de la Figura 1:

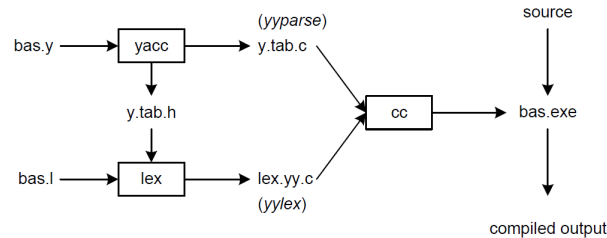


Figure 1: Construir un compilador/traductor usando Lex & Yacc (Niemann)

Para poder implementar este intérprete es necesario especificar todas las reglas de coincidencia de los patrones para lex (<archivo\_lex>.l) y las reglas de gramática para yacc (<archivo\_yacc>.y). Los comandos que necesitamos para crear nuestro compilador (a.out) se listan a continuación en la Figura 2:

```

yacc -d bas.y          # create y.tab.h, y.tab.c
lex bas.l              # create lex.yy.c
cc lex.yy.c y.tab.c -obas.exe # compile/link
  
```

Figure 2: Comandos para crear el compilador (Niemann)

**Yacc** lee las descripciones gramaticales contenidas en el archivo de extensión .y y genera un analizador de sintaxis (Analizador Sintáctico), que incluye la función yyparse, en el archivo de salida que este genera: y.tab.c. Incluido en el archivo .y hay declaraciones de tokens.

La opción `-d` (ver Figura 2) hace que yacc genere definiciones para los tokens y los coloque en el archivo de salida *y.tab.h*. Por otra parte Lex, lee las descripciones de los patrones en el archivo .l, incluyendo el archivo *y.tab.h* y genera un analizador léxico, que incluye la función yylex, en el archivo de salida generado al compilar con lex, esto es en: *lex.yy.c*.

Finalmente estos dos (lexer y el analizador) se compilan y vinculan entre sí para crear el archivo ejecutable **a.out**. Desde la función main llamamos a la función yyparse para ejecutar el compilador. La función yyparse llama automáticamente a yylex para obtener los tokens.

### 1.2.1 Acerca de la función *yywrap()*

En (Simmross) podemos encontrar una definición acertada sobre la función *yywrap()*, está se muestra a continuación:

*yywrap()* es una función que se utiliza para decidir qué debe hacer *yylex()* cuando se alcanza el final del fichero de entrada. Las opciones posibles son:

1. que *yylex()* finalice y devuelva el control a la función llamante (normalmente *main()*). Esto ocurre cuando *yywrap()* devuelve 1.
2. que *yylex()* continúe procesando la entrada de *yyin* como si no hubiera sucedido nada. Esto ocurre cuando *yywrap()* devuelve 0. Para que esto funcione, *yywrap()* debe llevar a cabo las acciones necesarias para que *yyin* apunte a alguna entrada que seguir procesando.

La función *yywrap()* que existe por defecto devuelve siempre 1, por eso *yylex()* finaliza en cuanto acaba de procesar un fichero de entrada. Si queremos construir un analizador que procese varios ficheros, uno detrás de otro, la manera idónea es proporcionar nuestra propia función *yywrap()* que:

1. Cierre el fichero de entrada actual (*yyin*).
2. Abra el siguiente fichero de entrada (con **fopen()**).
3. Apunte *yyin* al fichero recién abierto.
4. Devuelva 0.

*yylex()* seguirá procesando la entrada de *yyin* hasta que alcance el fin de fichero e *yywrap()* devuelva 1.

### 1.3 Acerca de la función *yylex()*

La función *yylex()* es el analizador léxico en sí. Devuelve 0 cuando se ha alcanzado el fin de fichero y el resultado de llamar a *yywrap()* ha sido 1. *yylex()* puede devolver otros valores si se incluyen sentencias **return** como parte de una acción dentro de la sección de reglas.

En (IBM) se nos muestra que la función de *yylex()*: *devuelve un valor que indica el tipo de token que se ha obtenido. Si el token tiene un valor real, este valor (o alguna representación del valor, por ejemplo, un puntero a una cadena que contiene el valor) se devuelve en una variable externa llamada **yyval***

## 1.4 Acerca de la función *yyparse()*

La función del analizador *yyparse()* es llamada para producir el análisis. En (Par) podemos encontrar que: *esta función lee tokens, ejecuta acciones y finalmente regresa cuando encuentra un fin de entrada o un error de sintaxis irrecoverable. También puede escribir una acción que indique yyparse regresar inmediatamente sin leer más.*

Algunas anotaciones importantes a las que podemos hacer referencia en (IBM) es que la función *yyparse()* tiene:

1. devuelve el valor de 0 si la entrada que analiza es válida de acuerdo con las reglas gramaticales dadas. Devuelve un 1 si la entrada es incorrecta y la recuperación de errores es imposible.
2. no hace su propio análisis léxico. En otras palabras, no separa la entrada en tokens listos para analizar. En cambio, llama a una rutina llamada *yylex()* cada vez que quiere obtener un token de la entrada.

## 2 Gramática de BASIC

Lo primero que debemos de hacer es definir la gramática expresada de la forma de *Backus-Naur*. Tomaremos dos ejemplos que se listan a continuación en las Figuras 3a y 3b:

```

line ::= number statement CR | statement CR
statement ::= PRINT expr-list
            IF expression relop expression THEN statement
            GOTO expression
            INPUT var-list
            LET var = expression
            GOSUB expression
            RETURN
            CLEAR
            LIST
            RUN
            END

expr-list ::= (string|expression) (, (string|expression) ) *
var-list  ::= var (, var) *
expression ::= (+|-|e) term ((+|-) term) *
term      ::= factor ((*|/) factor) *
factor    ::= var | number | (expression)
var       ::= A | B | C ... | Y | Z
number   ::= digit digit *
digit    ::= 0 | 1 | 2 | 3 | ... | 8 | 9
relop    ::= < | > | = | <= | >= | =
string   ::= " (a|b|c ... |x|y|z|A|B|C ... |x|y|z|digit) * "

```

(a) *Tiny BASIC grammar*

```

Unary_Op ::= - | !
Binary_Op ::= + | - | * | / | %
            = | < | > | <= | >= | <>
            & | ' | '
Expression ::= integer
            variable
            "string"
            Unary_Op Expression
            Expression Binary_Op Expression
            ( Expression )
Command ::= REM string
            GOTO integer
            LET variable = Expression
            PRINT Expression
            INPUT variable
            IF Expression THEN integer

Line ::= integer Command

Program ::= Line
          | Line Program

Phrase ::= Line | RUN | LIST | END

```

(b) *BASIC grammar*

La gramática de la Figura 3a se encuentra escrita en la forma de *Backus-Naur*<sup>1</sup>.

En la lista, un asterisco (\*) denota cero o más veces del objeto a su izquierda, excepto el primer asterisco en la definición de *término*. Como es común en la notación gramatical del lenguaje de computadora, la barra vertical (|) distingue alternativas, al igual que su listado en líneas separadas. El símbolo CR denota un retorno de carro (generalmente generado por la tecla **Enter** de un teclado) ([Wik](#)).

La siguiente gramática que usaremos de ejemplo se muestra a en la Figura 3b<sup>2</sup>.

Podemos ver que la forma en que se definen las expresiones no garantiza que se pueda evaluar una expresión bien formada. Por ejemplo, `1+ hola` es una expresión y, sin embargo, no es posible evaluarla. Esta elección deliberada nos permite simplificar tanto la sintaxis abstracta como el análisis del lenguaje básico. El precio a pagar por esta elección es que un programa básico sintácticamente correcto puede generar un error de tiempo de ejecución debido a una falta de coincidencia de tipos ([Christian.Queinnec@lip6.fr](#)).

<sup>1</sup>La gramática puede ser vista en: [https://en.wikipedia.org/wiki/Tiny\\_BASIC](https://en.wikipedia.org/wiki/Tiny_BASIC)

<sup>2</sup>Esta gramática puede ser vista en: <https://caml.inria.fr/pub/docs/oreilly-book/html/book-ora058.html>

### 3 Construcción del Analizador Sintáctico

Usando las gramáticas vistas en las Figuras 3a y 3b construiremos el analizador sintáctico correspondiente.

La gramática que implementaremos para construir el analizador sintáctico se muestra en la Tabla 1<sup>3</sup>.

Ahora bien, construiremos el analizador sintáctico implementando **GNU BISON**<sup>4</sup>.

El analizador sintáctico para la gramática propuesta en la Tabla 1 es el siguiente:

```

1  %{
2  #include <stdio.h>
3  #include <stdlib.h>
4  #include <stdarg.h>
5
6  int yylex();
7  int yyparse();
8  FILE *yyin;
9  int line_num;
10
11 void yyerror(const char *s);
12 %}
13
14 %token PRINT IF THEN GOTO INPUT LET GOSUB RETURN CLEAR LIST RUN END
15      STRING COMMA PLUMIN MULDIV EQ LEQGEQ CR DIGIT VAR LPAREN RPAREN
16 %%
17
18 program: lines    {printf( Successfull parsing of Tiny Basic program );}
19          ;
20
21 lines: lines line {printf( Successfully parsed lines -> lines line );}
22       | line {printf( Successfully parsed lines -> line );}
23       ;
24
25 line: number statement CR {printf( Successfully parsed line -> number statement
26      CR );}
27     | statement CR {printf( Successfully parsed line -> statement CR );}
28     ;
29 statement:
30     PRINT expr-list {printf( Parsed PRINT statement );}
31     | IF expression LEQGEQ expression THEN statement {printf( Parsed IF
      Statement );}

```

<sup>3</sup>La tabla no es una gramática fiel que se utilizara en los archivos finales, por lo que este modelo no servirá de guía para implementar el analizador sintáctico de *TinyBasic*

<sup>4</sup>GNU bison es un programa generador de analizadores sintácticos de propósito general perteneciente al proyecto GNU disponible para prácticamente todos los sistemas operativos, se usa normalmente acompañado de flex aunque los analizadores léxicos se pueden también obtener de otras formas.

```

32 | IF expression EQ expression THEN statement {printf( Parsed IF
    Statement );}
33 | GOTO expression {printf( Parsed GOTO statement );}
34 | INPUT var-list {printf( Parsed INPUT Statement );}
35 | LET VAR EQ expression {printf( Parsed LET Statement );}
36 | GOSUB expression {printf( Parsed GOSUB Statement );}
37 | RETURN {printf( Parsed RETURN Statement );}
38 | CLEAR {printf( Parsed CLEAR Statement );}
39 | LIST {printf( Parsed LIST Statement );}
40 | RUN {printf( Parsed RUN Statement );}
41 | END {printf( Parsed END Statement );}
42 ;
43
44 expr-list: expr-list COMMA expr-id {printf( Parsed expr-list -> expr-list
    COMMA expr-id );}
45 | expr-id {printf( Parsed expr-list -> expr-id );}
46 ;
47
48 expr-id: STRING {printf( Parsed expr-id -> STRING );}
49 | expression {printf( Parsed expr-id -> expression );}
50 ;
51
52 var-list: VAR COMMA var-list {printf( Parsed var-list -> VAR COMMA var-list );}
53 | VAR {printf( Parsed var-list -> VAR );}
54 ;
55
56 expression: expression PLUMIN term {printf( parsed expression -> expression
    PLUMIN term );}
57 | PLUMIN term {printf( Parsed expression -> PLUMIN term );}
58 ;
59
60 term: term MULDIV factor {printf( parsed term -> term MULDIV factor );}
61 | factor {printf( Parsed term -> factor );}
62 ;
63
64 factor: VAR {printf( parsed factor -> VAR );}
65 | number {printf( parsed factor -> number );}
66 | LPAREN expression RPAREN {printf( Parsed factor -> LPAREN expression
    RPAREN );}
67 ;
68
69 number: number DIGIT | DIGIT {printf( Parsed number );}
70 ;
71
72 %%
73
74 int main(int n, char** f) {
75     FILE *myfile;
76     if(n > 1) {
77         myfile = fopen(f[1], r);
78     } else {

```

```
79     printf( No se ha ingresado algun archivo );
80     exit(-1);
81 }
82
83 if(!myfile) {
84     printf( No se puede abrir el archivo! );
85     return -1;
86 }
87
88 yyin = myfile;
89
90 do {
91     yyparse();
92 } while(!feof(yyin));
93 }
94
95 void yyerror(const char *s) {
96     printf( Parse Error! Message on Line number: exit(-1);
```

Listing 1: Analizador Sintáctico

Como se puede observar en el código, la tabla de la gramática que declaramos anteriormente (ver Tabla 1) no fue implementada acorde a la derivación que se hizo. Se hicieron algunos cambios a la gramática, pero la idea es la misma.

## 4 Construcción del Analizador Léxico

El analizador léxico que corresponde a la gramática implementada en el Código ?? es el siguiente:

```
1  %{
2  #include <stdlib.h>
3  #define YY_DECL int yylex()
4  #include basicTiny.tab.h
5
6  int line_num = 1;
7  %}
8
9  %%
10
11 \      ;
12 PRINT {return PRINT;}
13 IF {return IF;}
14 THEN {return THEN;}
15 GOTO {return GOTO;}
16 INPUT {return INPUT;}
17 LET {return LET;}
18 GOSUB {return GOSUB;}
19 RETURN {return RETURN;}
```



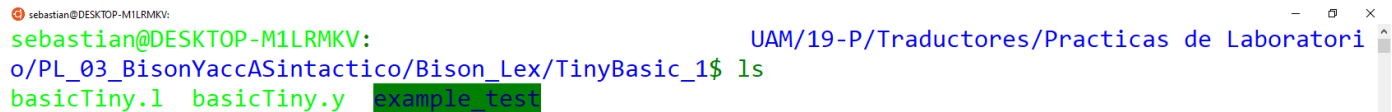
```
20 CLEAR {return CLEAR;}
21 LIST {return LIST;}
22 RUN {return RUN;}
23 END {return END;}
24 [\r\n] {++line_num; return CR;}
25 [<>] {return LEQGEQ;}
26 = {return EQ;}
27 [0-9] {return DIGIT;}
28 [\*/] {return MULDIV;}
29 [\+\-] {return PLUMIN;}
30 [A-Z] {return VAR;}
31 , {return COMMA;}
32 \( [a-z 0-9A-Z] ) + return STRING; return LPAREN; return RPAREN;
```

Listing 2: Analizador Léxico

## 5 Método de Compilación

Para compilar los archivos 1 y 2 siga los siguientes pasos:

1. En el directorio que se encuentran los archivos 1 y 2 abra una terminal (Linux ó Windows) como se muestra a continuación:



```
sebastian@DESKTOP-M1LRMKV: UAM/19-P/Traductores/Practicas de Laboratorio/PL_03_BisonYaccASintactico/Bison_Lex/TinyBasic_1$ ls
basicTiny.l  basicTiny.y  example test
```

Figure 4: Terminal de Linux en el directorio de archivos fuente

2. Ahora ya que estamos dentro de la carpeta en la terminal, escribiremos los siguientes comandos:

```
1 // Compilamos el archivo BISON el cual
2 // contiene el analizador sintactico
3
4 bison -d basicTiny.y
5
6 // Compilamos el archivo FLEX el cual
7 // contiene el analizador lexico
8
9 flex basicTiny.l
10
11 // Compilamos los archivos que se
12 // generaron a partir de BISON y FLEX/LEX
13
14 gcc basicTiny.tab.c lex.yy.c -lfl -o tinyBasicSP
15
16 // Para ejecutar el archivo ya compilado
17
18 ./tinyBasicSP
19
```

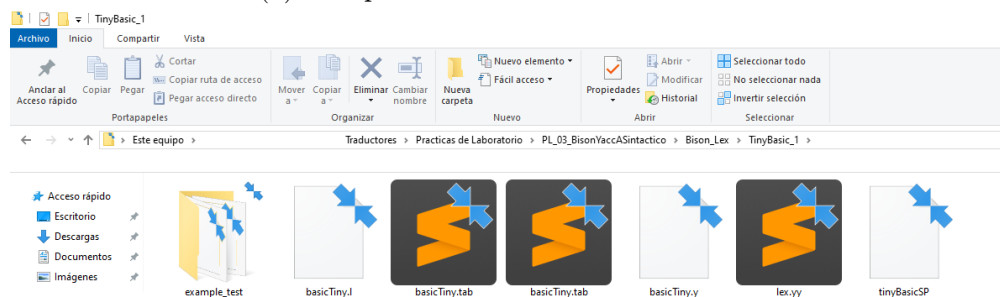
El resultado de haber compilado estos archivos se muestra a continuación:

```

sebastian@DESKTOP-M1LRMKV: /UAM/19-P/Traductores/Practicas de Laboratori
o/PL_03_BisonYaccASintactico/Bison_Lex/TinyBasic_1$ ls
basicTiny.l  basicTiny.y  example_test
sebastian@DESKTOP-M1LRMKV: /UAM/19-P/Traductores/Practicas de Laboratori
o/PL_03_BisonYaccASintactico/Bison_Lex/TinyBasic_1$ bison -d basicTiny.y
sebastian@DESKTOP-M1LRMKV: /UAM/19-P/Traductores/Practicas de Laboratori
o/PL_03_BisonYaccASintactico/Bison_Lex/TinyBasic_1$ flex basicTiny.l
sebastian@DESKTOP-M1LRMKV: /UAM/19-P/Traductores/Practicas de Laboratori
o/PL_03_BisonYaccASintactico/Bison_Lex/TinyBasic_1$ g++ basicTiny.tab.c lex.yy.c -lfl -o tinyBasic
SP

```

(a) Compilando los archivos en terminal



(b) Archivos generados despues de haber compilado

- Como podemos observar, al compilar el comando `bison -d basicTiny.y` genero los archivos **basicTiny.tab.c** y **basicTiny.tab.h**. Al ejecutar el comando `flex basicTiny.l` genero el archivo de salida **lex.yy.c**. Por ultimo, si ejecutamos el comando `g++ basicTiny.tab.c lex.yy.c -lfl -o tinyBasicSP` este genera el archivo ya compilado bajo el nombre **tinyBasicSP**.

## 6 Pruebas

A continuación se presentarán algunas de las pruebas que se realizaron al analizador sintáctico para una gramática de *Tiny BASIC*. Estas pruebas se realizaron con los siguientes archivos de prueba:

```
1 1 IF +(A-B) < +(9*(-C+D)) THEN PRINT A MINUS B IS LESS THAN 9 TIMES D MINUS C
2 2 END
3 1 INPUT A,B,C
4 2 LET A = -32
5 3 LET B = +A - 4
6 4 LET C = +(A / B)
7 5 PRINT +A +B +C
8 6 END
9 PRINT WELCOME
10 GOTO +A
11 LET A = -367 + 5 / 7
12 RETURN
13 1 GOSUB +A +B
14 LIST
15 3 END
16 INPUT A,B
17 LET A = -3
18 LET B = -A
19 PRINT Value of A is , +A , Value of B is , +B
20 CLEAR
21 END
```

Listing 3: Archivo de Pueba 1

```
1 PRINT Welcome
2 10 LET I = +1
3 20 IF +I > +10 THEN GOTO +70
4 30 PRINT +I
5 LIST
6 50 LET I = +I + 1
7 60 GOSUB +20
8 LET J = +21
9 INPUT K
10 LET L = +J +K
11 END
```

Listing 4: Archivo de Prueba 2

Si ejecutamos el archivo compilado **tinyBasicSP** junto a los archivos de prueba obtendremos la siguiente salida en consola:

```
sebastian@DESKTOP-M1LRMKV: /UAM/19-P/Traductores/Practicas de Laboratorio/PL_03_BisonYaccASintactico/Bison_Lex/TinyBasic_1
3_BisonYaccASintactico/Bison_Lex/TinyBasic_1$ ./tinyBasicSP example_test/test1.tb
Parsed number
parsed factor -> VAR
Parsed term -> factor
Parsed expression -> PLUMIN term
parsed factor -> VAR
Parsed term -> factor
parsed expression -> expression PLUMIN term
Parsed factor -> LPAREN expression RPAREN
Parsed term -> factor
Parsed expression -> PLUMIN term
Parsed number
parsed factor -> number
Parsed term -> factor
parsed factor -> VAR
Parsed term -> factor
Parsed expression -> PLUMIN term
parsed factor -> VAR
Parsed term -> factor
parsed expression -> expression PLUMIN term
Parsed factor -> LPAREN expression RPAREN
parsed term -> term MULDIV factor
Parsed expression -> PLUMIN term
Parsed factor -> LPAREN expression RPAREN
Parsed term -> factor
Parsed expression -> PLUMIN term
Parsed expr-id -> STRING
Parsed expr-list -> expr-id
Parsed PRINT statement
Parsed IF Statement
Successfully parsed line -> number statement CR
Successfully parsed lines -> line
Successfull parsing of Tiny Basic program
Parse Error! Message on Line number 3 :: syntax error
```

Figure 6: Salida en consola al ejecutar el Analizador Sintáctico

## 7 Conclusiones

El analizador sintáctico construido a partir de la guía de la Tabla 1 funciono de manera exitosa. Esto puede comprobarse en la salida que se obtiene al ejecutar el archivo **tinyBasicSP** con un ejemplo básico (ver Ejemplo de Prueba 3 ó 4).

En la Figura 6 podemos darnos cuenta de que se va creando una especie de árbol como los desarrollados en clase.

La realización de está práctica no fue nada sencilla, ya que para definir con que gramática se trabajaría se tuvieron que hacer muchos cambios, incluso se descartaron las gramáticas de las Figuras 3a y 3b. Por último queda pendiente la parte del desarrollo en CUP (Java), pero por motivo de tiempo se ha decidido no anexarlo a este reporte.

Los códigos se van a alojar en una carpeta de Drive, con la siguiente liga podrá acceder a los archivos fuente y a este documento. Presiona en el siguiente [link](#) para acceder a la carpeta compartida.

## 8 Anexo

Gramática para Tiny BASIC
prog := block
block := block line   line
line := INTEGER statement CR   statement CR
statement := PRINT expr_list   IF expression relop expression THEN statement   GOTO expression   INPUT var_list   LET var = expression   END
expr_list := expr_list, expression   expression
var-list := var-list , var   var
expression := expression + term   expression - term   term
term := term * factor   term / factor   factor
factor := var   number   (expression)
number := INTEGER   DECIMAL
var := A   B   C ....   Y   Z
relop := <   <=   >   >=   ==   !=

Table 1: Tabla de Gramatica para el A. Sintáctico

## References

[Par] The parser function `yyparse`.

[Wik] Tiny basic.

[IBM] `yyparse()` and `yylex()`.

[Christian.Queinnec@lip6.fr] Christian.Queinnec@lip6.fr.

[Niemann] Niemann, T. *Lex Yacc Tutorial*. Portland, Oregon.

[Simmross] Simmross, F. El generador de analizadores léxicos `lex`.  
<https://www.infor.uva.es/mluisa/talf/docs/labo/L6.pdf>.