



Instituto Politécnico Nacional
Escuela Superior de Cómputo



❖ **UNIDAD DE APRENDIZAJE:** COMPILADORES

❖ **PROFESOR:** RAFAEL NORMAN SAUCEDO DELGADO

❖ **EQUIPO:**

ARTEAGA LARA SAMUEL
BARRERA PÉREZ TONATIHU
FLORES TEPATL GISELLE
MENDOZA JAIMES IAN
MONROY MARTOS ELIOTH

❖ **Práctica 8. Generación de un analizador sintáctico
ascendente para la gramática de un lenguaje de
programación (HOC)**

❖ **GRUPO:** 3CM6

Introducción

Bison es un generador de analizadores sintácticos de propósito general que convierte una descripción para una gramática independiente del contexto (en realidad de una subclase de éstas, las LALR) en un programa en C que analiza esa gramática. Es compatible al 100% con Yacc, una herramienta clásica de Unix para la generación de analizadores léxicos, pero es un desarrollo diferente realizado por GNU bajo licencia GPL. Todas la gramáticas escritas apropiadamente para Yacc deberían funcionar con Bison sin ningún cambio. Usándolo junto a Flex esta herramienta permite construir compiladores de lenguajes.

Un fuente de Bison (normalmente un fichero con extensión .y) describe una gramática. El ejecutable que se genera indica si un fichero de entrada dado pertenece o no al lenguaje generado por esa gramática. La forma general de una gramática de Bison es la siguiente:

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

Los '%%', '%{' y '%}' son signos de puntuación que aparecen en todo archivo de gramática de Bison para separar las secciones.

Las declaraciones en C pueden 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 son las producciones de la gramática, que además pueden llevar asociadas acciones, código en C, que se ejecutan cuando el analizador encuentra las reglas correspondientes.

El código C adicional puede contener cualquier código C que desee utilizar. A menudo suele ir la definición del analizador léxico yylex, más subrutinas invocadas

por las acciones en las reglas gramaticales. En un programa simple, todo el resto del programa puede ir aquí.

Desarrollo

La gramática vinculada con el lenguaje de programación HOC se encuentra en el libro *The Unix programming environment*, esta se encuentra en forma de código para bison, sin embargo dicho código incluye un análisis semántico y debido a que esa parte no fue necesaria en esta práctica es por esto que los siguientes archivos de bison contienen código que sólo realiza un análisis sintáctico; dicho análisis consiste en la impresión de las acciones que se realizaron con cada producción de la gramática.

El siguiente archivo es el que utiliza flex para realizar el análisis léxico, el archivo contiene las expresiones regulares que tiene el lenguaje de programación HOC.

hoc.l

```
%{
    #include "hoc.tab.h"
    #include <stdlib.h>
}%

%%

"PI" { return PI; }
"if"  { return IF; }
"else" { return ELSE; }
"while" { return WHILE; }
"return" { return RETURN; }

">" { return GT; }
"<" { return LT; }
"!=" { return NE; }
"==" { return EQ; }
">=" { return GE; }
"<=" { return LE; }

/* FUNCIONES */
"sqr" { return FUNCTION; }
"log" { return FUNCTION; }
"print" { return PRINT; }
"bltin" { return BLTIN; }
"read" { return READ; }
```

```

"func" {return FUNC;}
"proc" {return PROC;}
[a-zA-Z$][a-zA-Z$0-9]* {return VAR;}

\"^[^\\n]*\" { return STRING;}

[0-9]+ {yyval.NUM = (double) atoi(yytext); return NUM;}
'-' {return NEG;}
[ \\t]+
. {return *yytext;}
\\n {return '\\n';}

%%

```

Para el análisis sintáctico se tiene el siguiente archivo cuyo contenido es las reglas de producción de la gramática del lenguaje y lo que se realiza con cada producción.

hoc.y

```

%{
    #include <stdio.h> /* For printf, etc. */
    #include <stdlib.h>
    #include <math.h> /* For pow, used in the grammar. */
    int yylex (void);
    void yyerror (char const *);
    extern FILE *yyin;
}%

#define api.value.type union /* Generate YYSTYPE from these types: */
%token <double> NUM PRINT STRING PI /* Simple double precision number. */
%token <double> READ BLTIN PROCEDURE PROC VAR FUNC FUNCTION ELSE WHILE IF
RETURN /* Symbol table pointer: variable and function. */
%type <double> expr asig

%right '='
%left OR
%left AND
%left GT GE LT LE EQ NE
%left '-' '+'
%left '*' '/'
%left NEG NOT /* negation--unary minus */
%right '^' /* exponentiation */

%% /* The grammar follows. */
lista: %empty

```

```

| lista '\n'
| lista defunc '\n'      { printf("lista-defunc\n"); }
| lista asig '\n'
| lista stmt '\n'       { printf("lista-stmt\n"); }
| lista expr '\n'       { printf("lista-expr %f\n", $2); }
| lista error '\n'      { yyerrok; }
;

stmt: expr                { printf("stmt-expr\n"); }
| RETURN
| RETURN expr
| PROCEDURE '(' arglist ')'
| PRINT prlist           { printf("print\n"); }
| while condicion stmt end { printf("Vi un while\n"); }
| if condicion stmt end   { printf("Vi un if\n"); }
| if condicion stmt end ELSE stmt end { printf("Vi un else\n"); }
| '{' stmtlist '}'
;

stmtlist: %empty
| stmtlist '\n'
| stmtlist stmt
;

expr: NUM                { printf("Vi un numero\n"); }
| VAR                   { printf("Vi una variable\n"); }
| PI
| asig
| READ '(' VAR ')'
| BLTIN '(' expr ')'
| '(' expr ')'          { printf("parentesis expr\n"); }
| expr '+' expr
| expr '-' expr         { printf("Vi un menos\n"); }
| expr '*' expr         { printf("Vi una multiplicacion\n"); }
| expr '/' expr         { printf("Vi una division\n"); }
| expr '^' expr         { printf("Vi una potencia\n"); }
| '-' expr %prec NEG    { printf("Vi un menos\n"); }
| expr GT expr          { printf("Vi un GT\n"); }
| expr GE expr          { printf("Vi un GE\n"); }
| expr LT expr          { printf("Vi un LT\n"); }
| expr LE expr          { printf("Vi un LE\n"); }
| expr '+' expr
| expr '-' expr
| expr '*' expr
| expr '/' expr
| expr '^' expr
| '-' expr %prec NEG
| expr GT expr
| expr GE expr
| expr LT expr
| expr LE expr

```

```

| expr EQ expr      { printf("Vi un EQ\n"); }
| expr NE expr      { printf("Vi un NE\n"); }
| expr AND expr     { printf("Vi una AND\n"); }
| expr OR expr      { printf("Vi una OR\n"); }
| NOT expr          { printf("Vi una NOT\n"); }
| VAR '(' arglist ')' { printf("Vi una funcion\n"); }
| FUNCTION '(' arglist ')' { printf("Vi una funcion\n"); }
;

```

```

prlist: expr
| STRING
| prlist ',' expr
| prlist ',' STRING
;

```

```

arglist: %empty
| expr
| arglist ',' expr
;

```

```

asig: VAR '=' expr { printf("Vi una asignacion\n"); }
;

```

```

condicion: '(' expr ')' { printf("Vi una condicion\n"); }
;

```

```

while: WHILE
;

```

```

if: IF
;

```

```

end: %empty
;

```

```

defunc: FUNC procnombre { printf("Vi un FUNC\n"); }
      '(' ')' stmt { printf("stmt\n"); }
| PROC procnombre { printf("Vi un FUNC\n"); }
      '(' ')' stmt { printf("stmt\n"); }
;

```

```

procnombre: VAR { printf("VAR\n"); }
;

```

```

/* End of grammar. */

```

```

%%

/* Called by yyparse on error. */
void
yyerror (char const *s)
{
    fprintf (stderr, "%s\n", s);
}

int main (int argc, char const* argv[]) {
    if (argc < 2) {
        printf("Faltan parametros ./analizador <archivo>");
    }
    FILE *archivo = fopen(argv[1], "r");
    if (!archivo) {
        printf("No se inserto el archivo\n");
        return -1;
    }
    yyin = archivo;
    do {
        yyparse();
    } while(!feof(yyin));
    return 0;
}

```

Finalmente, es necesario compilar los archivos haciendo uso de algunas banderas, para facilitar el trabajo se creó un Makefile con los comandos más usados al elaborar esta práctica.

Makefile

```

run: analizador.out
    ./analizador.out
analizador.out: hoc.tab.o lex.yy.o
    gcc -lm -lfl lex.yy.o hoc.tab.o -o ./analizador.out
lex.yy.o: lex.yy.c
    gcc -c -lfl -Wall lex.yy.c -o lex.yy.o
lex.yy.c:
    flex hoc.l
hoc.tab.o: hoc.tab.c
    gcc -c -Wall hoc.tab.c -o hoc.tab.o
hoc.tab.c:
    bison -d -Wconflicts-sr hoc.y
lex:
    flex hoc.l

```

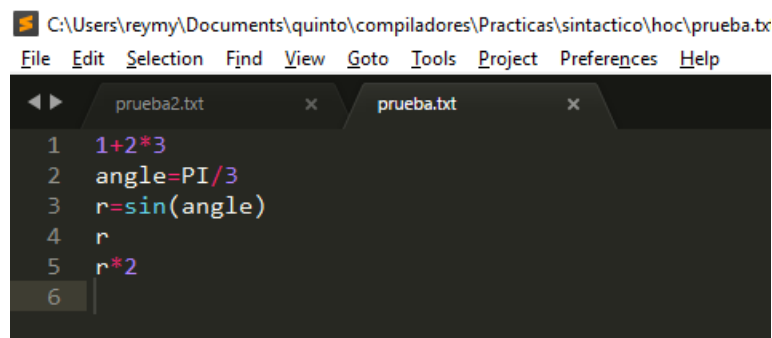
```
clean:
rm -f *.o ./analizador.out ./hoc.tab.* ./lex.yy.c
```

Resultados

Para poder observar los resultados de esta práctica se realizaron pruebas introduciendo diferentes archivos de texto que contienen código HOC y verificar si cada archivo es correcto tanto léxicamente como sintácticamente.

Prueba 1

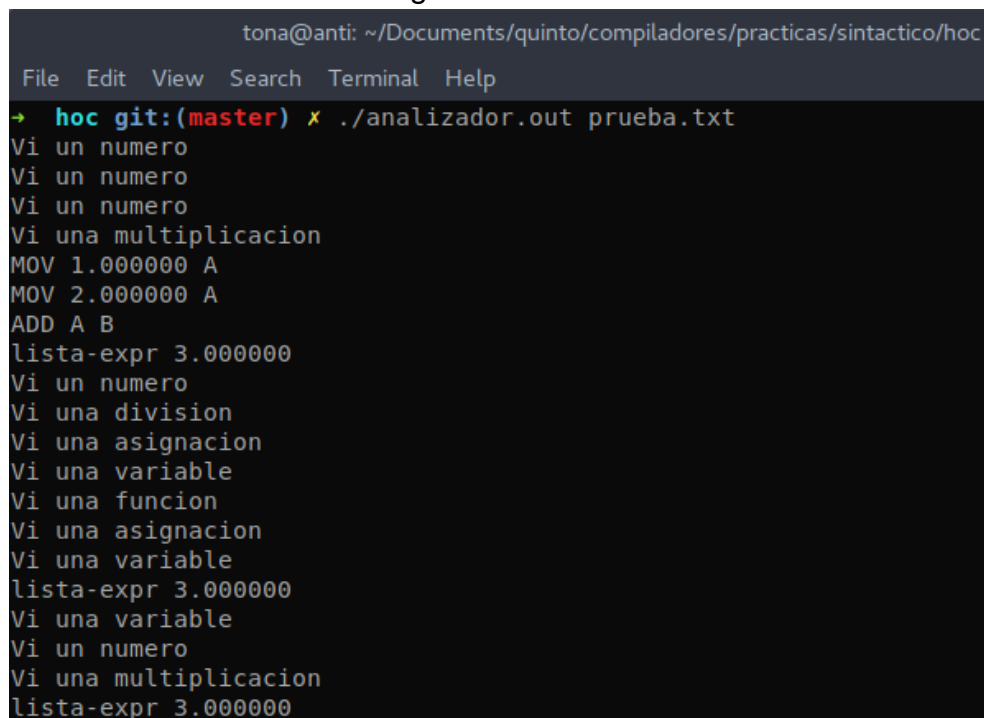
El archivo para esta prueba fue.



A screenshot of a text editor window titled 'C:\Users\reymy\Documents\quinto\compiladores\Practicas\sintactico\hoc\prueba.tx'. The editor has a menu bar with 'File', 'Edit', 'Selection', 'Find', 'View', 'Goto', 'Tools', 'Project', 'Preferences', and 'Help'. There are two tabs open: 'prueba2.txt' and 'prueba.txt'. The 'prueba.txt' tab is active, showing the following code:

```
1 1+2*3
2 angle=PI/3
3 r=sin(angle)
4 r
5 r*2
6
```

La respuesta del analizador fue la siguiente.



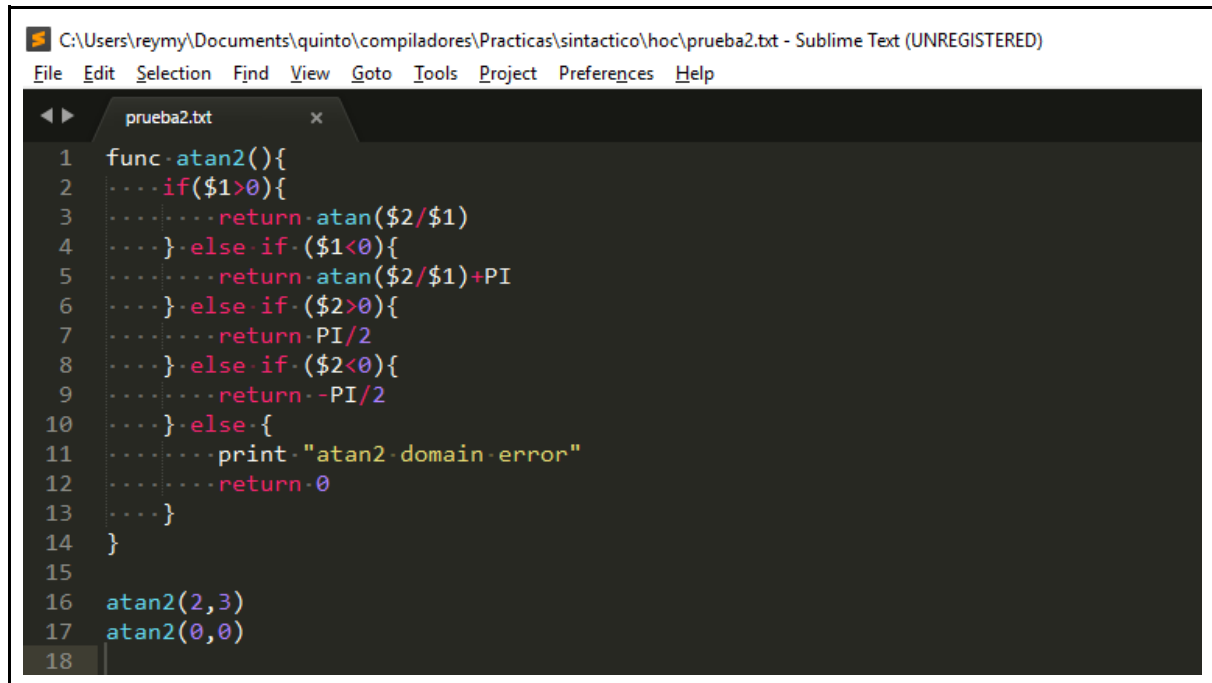
A screenshot of a terminal window with the title 'tona@anti: ~/Documents/quinto/compiladores/practicas/sintactico/hoc'. The terminal shows the command `./analizador.out prueba.txt` being executed. The output consists of a series of messages and assembly-like instructions:

```
→ hoc git:(master) x ./analizador.out prueba.txt
Vi un numero
Vi un numero
Vi un numero
Vi una multiplicacion
MOV 1.000000 A
MOV 2.000000 A
ADD A B
lista-expr 3.000000
Vi un numero
Vi una division
Vi una asignacion
Vi una variable
Vi una funcion
Vi una asignacion
Vi una variable
lista-expr 3.000000
Vi una variable
Vi un numero
Vi una multiplicacion
lista-expr 3.000000
```

Cada línea que se imprime es una instrucción a realizar despues de utilizar alguna regla de produccion de la gramática.

Prueba 2

El archivo para esta prueba fue.



```
C:\Users\reymy\Documents\quinto\compiladores\Practicas\sintactico\hoc\prueba2.txt - Sublime Text (UNREGISTERED)
File Edit Selection Find View Goto Tools Project Preferences Help

prueba2.txt x
1 func atan2(){
2     ...if($1>0){
3     ...return atan($2/$1)
4     ...} else if ($1<0){
5     ...return atan($2/$1)+PI
6     ...} else if ($2>0){
7     ...return PI/2
8     ...} else if ($2<0){
9     ...return -PI/2
10    ...} else {
11    ...print "atan2 domain error"
12    ...return 0
13    ...}
14 }
15
16 atan2(2,3)
17 atan2(0,0)
18
```

La respuesta del analizador fue la siguiente.

tona@anti: ~/Documents/quinto/compiladores/pr

File Edit View Search Terminal Help

+ hoc git:(master) x ./analizador.out prueba2.txt

VAR

Vi un FUNC

Vi una variable

Vi un numero

Vi un GT

Vi una condicion

Vi una variable

Vi una variable

Vi una division

Vi una funcion

Vi una variable

Vi un numero

Vi un LT

Vi una condicion

Vi una variable

Vi una variable

Vi una division

Vi una funcion

MOV 0.000000 A

MOV 0.000000 A

ADD A B

Vi una variable

Vi un numero

Vi un GT

Vi una condicion

Vi un numero

Vi una division

Vi una variable

Vi un numero

Vi un LT

Vi una condicion

Vi un menos

Vi un numero

Vi una division

print

Vi un numero

Vi un else

Vi un else

Vi un else

Vi un else

stmt

lista-defunc

Vi un numero

Vi un numero

Vi una funcion

lista-expr 0.000000

Vi un numero

Vi un numero

Vi una funcion

lista-expr 3.000000

+ hoc git:(master) x

Conclusiones

Arteaga Lara Samuel

El analizador sintáctico así como otros analizadores que componen a un compilador tiene una labor esencial y cuya implementación puede llegar a tomar bastante tiempo, sin embargo existen herramientas que con base en reglas gramaticales nos ayudan a la construcción de dicho analizador, en este caso Bison es una muestra inminente de la ayuda que aporta, sin embargo si es necesario tener conocimientos del tema, ya que su manejo no es tan simple o fácil, por medio de la realización de esta práctica pude adentrarme más en el manejo tanto de Flex como de Bison, lo cual fue una manera correcta de aprender a usar un poco de todo lo que ofrece, ya que ambas van de la mano y hasta cierto punto se complementan.

Barrera Pérez Carlos Tonatihu

Sin duda el uso de herramientas como bison facilitan el trabajo de realizar un analizador sintáctico ya que la mayor complicación que puede surgir es diseñar una gramática que pueda modelar el lenguaje que queremos sin ambigüedades que puedan producir errores en la creación del analizador sintáctico. Sin embargo, es importante mencionar que se debe de leer la documentación de bison ya que no es tan intuitivo de aprender, a su vez es importante tener conocimientos sobre flex ya que el uso de ambos va de la mano.

Flores Tepatl Giselle

Con la elaboración de esta práctica, pudimos hacer uso de Bison, un generador de analizadores sintácticos. La cual, resulta ser una herramienta que permite hacer de la construcción de un analizador sintáctico, un proceso más rápido y sencillo; y por ende, al ser una de las fases de un compilador, permite que la construcción del mismo sea más fácil; además por lo general, suele utilizarse junto con FLEX, el cual permite crear analizadores léxicos; de tal manera que la comunicación que existe entre estas 2 fases del compilador resulta práctica.

Mendoza Jaimes Ian

Los analizadores sintácticos se construyen a partir de gramáticas libres de contexto, el problema es que construir uno desde cero puede llevar bastante tiempo, por lo que los generadores de analizadores sintácticos se tornan una herramienta muy útil para la creación de compiladores de una manera rápida y eficiente. En particular Bison es un generador practico y facil de descargar que puede integrarse con Lex lo que lo vuelve una buena opción para diseñar un compilador.

Monroy Matos Elioth

El uso de los generadores de analizadores sintácticos son una herramienta muy útil para la construcción de compiladores, el usado en esta práctica, bison es una herramienta que permite crear el analizador sintáctico a partir de especificar las reglas gramaticales que queremos que este verifique que los tokens arrojados por el analizador léxico cumplan con las reglas definidas. En esta práctica combinamos el uso de flex y de bison para así tener un analizador léxico y un analizador sintáctico los cuales pudieran analizar la entrada y trabajar en conjunto. A pesar de que flex y bison son herramientas que facilitan el trabajo, es necesario revisar la documentación y principalmente ejemplos de uso de estos, ya que es un tanto complicado entender como trabajar con ellos al inicio.

Referencias

- Aho, A. (2007). *Compilers: Principles, Techniques, and Tools* (2nd ed.). Boston: Pearson/Addison Wesley.
- Kernighan, B., & Pike, R. (1984). *The Unix programming environment* (1st ed.). Englewood Cliffs, N.J.: Prentice-Hall.
- Levine, J. (2009). *Flex & bison* (1st ed.). Sebastopol, Calif.: O'Reilly Media.