Universidad Nacional Autónoma de México
Facultad de Ingeniería.

Nombre del profesor: Castañeda Castañeda Manuel Enrique
Grupo: 05

Proyecto de Compiladores

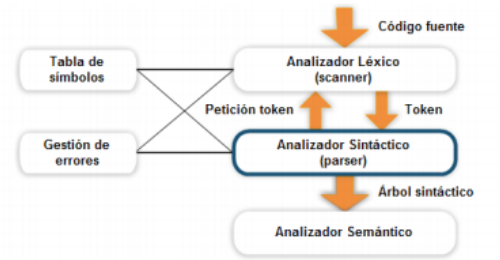
Integrantes:
Cruz Matias Yuridia Elizabeth
González González Héctor Emilio
Hernández Guzmán Marian Lisette

Fecha entrega del reporte
12 / Agosto / 2021

Índice

Introducción

Las fases de la compilación, entre las que se encontraba la fase de análisis sintáctico, son las siguientes:



El analizador sintáctico obtiene una cadena de tokens del analizador léxico y genera un árbol sintáctico, siendo el objetivo de esta fase la de comprobar que la secuencia de componentes léxicos que le entrega el analizador léxico cumple las reglas de la gramática que se han definido previamente.

Hay dos categorías generales de algoritmos para construir analizadores sintácticos: **los analizadores sintácticos descendentes y los analizadores sintácticos ascendentes.** Estos a su vez se subdividen en varios métodos, cada uno de los cuales tiene distintas capacidades y propiedades.

Análisis sintáctico ascendente

El **análisis sintáctico ascendente** es una técnica de análisis sintáctico que intenta comprobar si una cadena x pertenece al lenguaje definido por una gramática L (G) aplicando los siguientes criterios

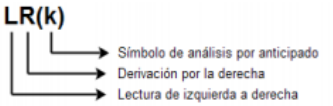
- Partir de los elementos terminales de la frase x
- Escoger reglas gramaticales estratégicamente
- Aplicar a la inversa derivaciones por la derecha (Right Most Derivation)
- Procesar la cadena de izquierda a derecha › Intentar alcanzar el axioma para obtener el árbol de análisis sintáctico o error

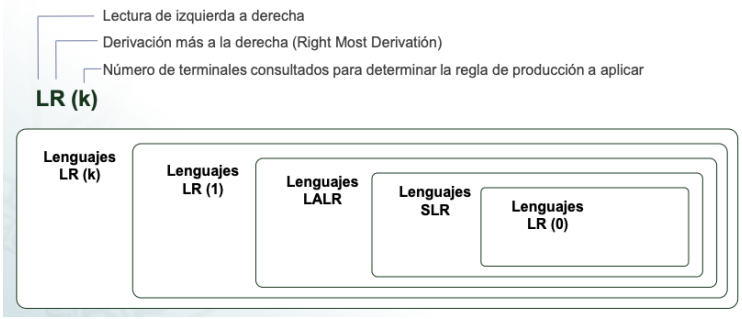
Existen dos tipos de analizadores ascendentes:
Analizadores de precedencia de operador: Se utiliza para un pequeño conjunto de gramáticas, denominadas gramáticas de operadores (una propiedad que deben cumplir estas gramáticas es que no pueden tener dos terminales seguidos). Se definen relaciones de precedencia disjuntos entre los terminales, que son los operadores, y estas relaciones guían la selección de las producciones.
Analizadores LR: Hacen un examen de la entrada de izquierda a derecha (left-to right, L) y construyen las derivaciones por la producción más a la derecha (rightmost derivation, R). Estos analizadores también son denominados analizadores por desplazamiento y

reducción. Hay varios autómatas LR y todos llevan un símbolo de análisis por anticipado de la entrada (símbolo de lookahead).

Los analizadores sintácticos ascendentes son capaces de decidir qué regla de producción aplicar a cada paso en función de los elementos terminales que se encuentran en la cabeza de lectura de la cadena de entrada. Como consecuencia se consigue un proceso de análisis con complejidad lineal O (n) con respecto al tamaño del problema. Estos analizadores son llamados genéricamente analizadores LR (K) o analizadores por reducción desplazamiento.

Hay varios tipos de analizadores sintácticos LR, dependiendo de cómo se construya el AFD y la tabla de análisis en el que están basados. El significado del acrónimo LR(k) es el siguiente:



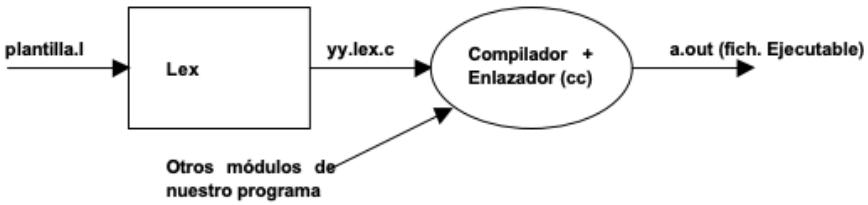


LEX

Lex es una herramienta de los sistemas UNIX/Linux que nos va a permitir generar código C que luego podremos compilar y enlazar con nuestro programa.

La principal característica de Lex es que nos va a permitir asociar acciones descritas en C, a la localización de las Expresiones Regulares que le hayamos definido. Para ello Lex se apoya en una plantilla que recibe como parámetro, y que deberemos diseñar con cuidado.

Internamente Lex va a actuar como un autómata que localizará las expresiones regulares que le describamos, y una vez reconocida la cadena representada por dicha expresión regular, ejecutará el código asociado a esa regla. Externamente podemos ver a Lex como una caja negra con la siguiente estructura:

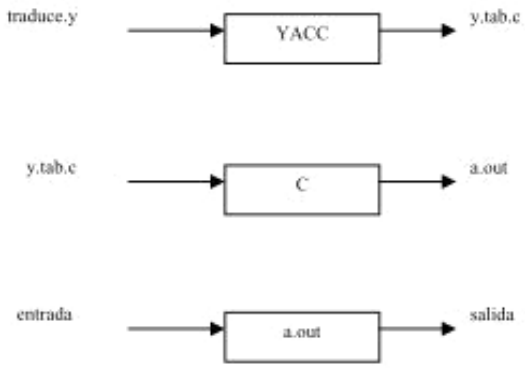


YACC

Llamado genéricamente así debido a que sus siglas provienen del inglés Yet Another Compiler Compiler un programa que parte del archivo escrito en el lenguaje específico Lenguaje de Descripción de Gramática y genera el código en C, C++ o Pascal. Permite un lenguaje propio para automatizar tareas repetitivas que, de otra manera, tendrían que hacerse manualmente, dilapidando mucho tiempo y subutilizando las posibilidades de las computadoras actuales

El archivo que se le da al Yacc par que éste lo procese y genere en el lenguaje fuente deseado está escrito con un diseño para tal fin, y es bastante sencillo, pues no posee un propósito general. Pero mantiene la potencia del lenguaje de salida que se utilice, pues todas las construcciones válidas aquí pueden escribirse si se utiliza una serie de convenios rigurosamente especificados. Este archivo se denomina de entrada, mientras que al generado por el Yacc es el de salida al de la fuente en que es escrito.

El archivo de entrada al Yacc consta de 2 partes fundamentales. La primera es un encabezamiento donde se definen los tipos de datos con los que se va a trabajar y se declaran los tokens, o palabras del lenguaje que se define. En la segunda se especifican todas las reglas que definen de forma rigurosa y unívoca la gramática del lenguaje que se desea interpretar. Además, se precisa qué es lo que se realiza a cada uno de los casos posibles.



Desarrollo

Elaborar analizador sintáctico ascendente que revise las sentencias de acuerdo a la siguiente gramática:

E→ T
E→E+T
E→E-T
T→i
T→(E)

i – Será cualquier número real o identificador

La cadena de tokens a revisar será entregada en un archivo por un analizador léxico hecho en lex

El analizador léxico reconocerá: números reales, identificadores, símbolos de operación (+,-) y los paréntesis (,)

Ejemplo de cadena de tokens que entrega el AL

i-(i+i)

- El ASA recibe el archivo con las cadenas de tokens separadas por ;
- Deberá indicar si es cadena valida
- Deberá indicar en que líneas se encuentran los posibles errores

Trabajamos con la gramática aplicando los siguiente algoritmos:

Gramática	**Factorizando**	**Quitando recursividad**
$E \rightarrow T$	$\propto = E$	$\propto = E'$
$E \rightarrow E + T$	$\beta 1 = +T$	$\beta = T$
$E \rightarrow E - T$	$\beta 2 = -T$	
$T \rightarrow i$		$E \rightarrow TE''$
$T \rightarrow (E)$	$E \rightarrow T$	$E'' \rightarrow E'E''$
	$E \rightarrow EE'$	$E'' \rightarrow$
	$E' \rightarrow +T$	$E' \rightarrow +T$
	$E' \rightarrow -T$	$E' \rightarrow -T$

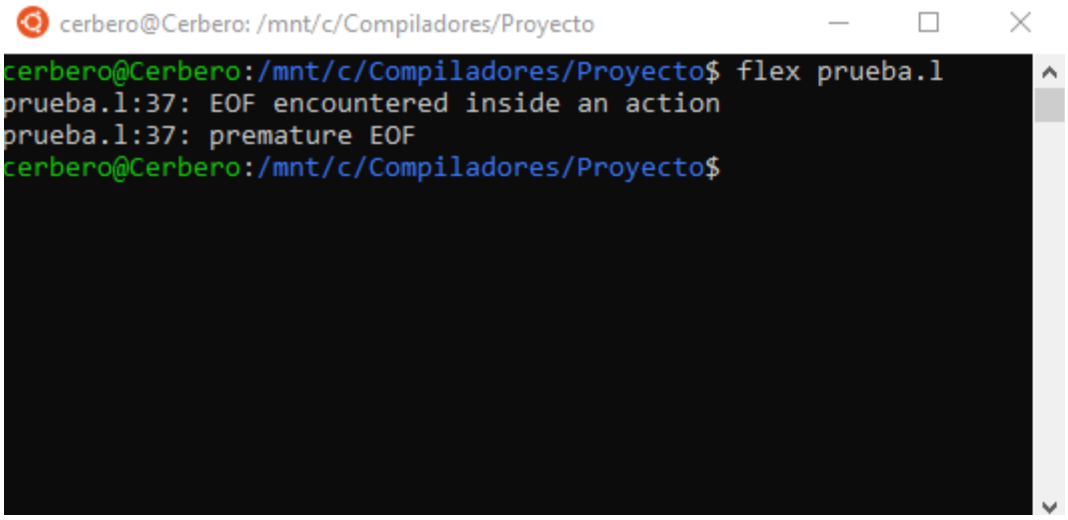
$T \rightarrow i$ $T \rightarrow i$

$T \rightarrow (E)$ $T \rightarrow (E)$

Con ayuda de la factorización y quitando la recursividad se intentó realizar el proyecto por medio de un único archivo lex obteniendo lo siguiente:

```
%{
#include <stdio.h>
%}

entero [0-9]
real {entero}*\.?{entero}*
letra [a-zA-Z]*
id {letra}*
operadores "+"|"-"
parIzq "("
parDer ")"

/*  E' = A
/*  E'' = B

E {T}{B}
A "+"{T}|"-"{T}
B {A}{B}
T {real}|{id}|"("{E}")"

%option noyywrap
%option yylineno
%%

{E}     {printf("Palabra aceptada\n");}
"\n"    {yylineno;}
.       {printf("\n\t Error linea %d\nCadena no reconocida %s\n",yylineno,yytext);}

%%
main(int argc,char *argv[]){
    if (argc > 1)
        yyin = fopen(argv[1],"r");
    else
        printf("ERROR");
    yylex();
}
```

Pero obtenemos el siguiente resultado que se da porque estamos metiendo una recursividad que lex no puede detectar.



Bajo este problema se investigó y se decidió manejarlo con yacc (previamente explicado) que maneja analizadores sintácticos, con esto se obtuvo dos archivos, lex.l y yacc.y

En lex se declaran las expresiones regulares que crean nuestras cadenas, lex obtiene los tokens (cada cadena detectada) y se los envía a yacc por medio de un return.

En la primera sección únicamente están algunas sentencias de c lo destacable es el y.tab.h que es un archivo que se genera y que nos permitirá compilar

```
%{
    #include "y.tab.h"
    #include <stdio.h>
%}
```

En sección donde obtenemos algunas expresiones regulares necesarias para poder mandar los tokens y que se lean las cadenas

```
%option yylineno

entero [0-9]
real {entero}*\.?{entero}*
letra [a-zA-Z]*
id {letra}*
crlf "\n"
```

La ultima sección que declaramos nos ayuda a reconocer las cadenas y mandarlas en forma de tokens a yacc.

```
%%
{real}      {yylval.numero = atoi(yytext);
                return REAL;}
{id}        {strcpy(yylval.identificador, yytext);
                return ID;}
"+"|"-"     return OPE;
"("         return PARIZQ;
")"         return PARDER;
{crlf}      return CRLF;
[\t\r. ]  {}
%%
```

yacc.y

Yacc recibe los tokens y desde un archivo lee línea por línea para verificar que la cadena pertenece a la gramática

En la primera sección únicamente están algunas sentencias, podemos destacar que se llama yyerror el cual es el manejo de errores de yacc, tenemos la declaración del fichero para poder leer un archivo externo, por último, yylineno que viene desde lex y nos ayudara a saber en qué línea estamos ubicados.

```
%{
    #include <stdio.h>
    void yyerror(char *msg);
    int yylex(void);
    extern FILE *yyin;
    extern int yylineno;
%}
```

La declaración %union especifica la colección de posibles tipos de datos de yylval y de los atributos

```
%union {
    float numero;
    char identificador[100];
};
```

Declaramos los tokens de lex e inicializamos nuestra gramática con ayuda de start

```
%token <numero> REAL
%token <identificador> ID
%token OPE
%token PARIZQ
%token PARDER
%token CRLF

%start INICIO
```

Declaramos nuestra gramática, cabe mencionar que se declaro dos No terminales mas para poder aceptar o no la cadena y el salto de línea

```
%%
INICIO: GRAM GRAM | GRAM;
GRAM:   E CRLF {printf("Cadena Aceptada\n");}
        | error CRLF {yyerrok;};
E:      T
        | E OPE T;
T:      ID
        | REAL
        | PARIZQ E PARDER;

%%
```

Formulamos lo que nos imprimirá cuando una cadena no sea detectada

```
void yyerror(char *msg){
    printf("\tError Sintactico en linea %d\n",yylineno);
}
```

Código para abrir archivo y llama yyparse para que comience el análisis

```
void main(int argc, char *argv[]){
    if (argc >1)
        yyin=fopen(argv[1],"rt");
    else
        printf("ERROR\n");
    yyparse();
    fclose(yyin);
}
```

Análisis de resultados

Analizando el código paso a paso:

lex.l

C: > Compiladores > Proyecto > lex.l

```
%{ /*Seccion Declaraciones*/
    #include "y.tab.h"
    #include <stdio.h>
%}

%option yylineno

entero [0-9]
real {entero}*\.?{entero}*
letra [a-zA-Z]*
id {letra}*
crlf "\n"
/*Seccion Reglas*/
%%
{real}      {yylval.numero = atoi(yytext);
                return REAL;} /*convierte la cadena en un numero para que yacc lo reconozca*/
{id}        {strcpy(yylval.identificador, yytext);
                return ID;} /*Pone en la posicion identificador a la cadena para que recibilo yacc*/
"+"|"-"     return OPE;
"("         return PARIZQ;
")"         return PARDER;
{crlf}      return CRLF; /*Manda un salto de linea para que pueda leer la siguiente cadena*/
[\t\r. ]  {} /*No hace nada*/
%%
```

```
void main(int argc, char *argv[]){
    /*Abre el archivo*/
    if (argc >1)
        yyin=fopen(argv[1],"rt");
    else
        printf("ERROR\n");
    /*Llama a parse que realiza la gramatica*/
    yyparse();
    fclose(yyin);
}
```

yacc.y

C: > Compiladores > Proyecto > yacc.y

```
%{/*Seccion Declaraciones*/
    #include <stdio.h>
    void yyerror(char *msg); /*Manejo de errores en Yacc es necesario que
                               el usuario la proporcione*/
    int yylex(void); /*Necesario para hacer referencia a Lex*/
    extern FILE *yyin; /*Apuntador a Fichero*/
    extern int yylineno; /*Para poder leer las lineas heredadas de Lex*/
%}
%union { /*Union necesario para poder recibir las cadenas de lex*/
    float numero; /*recibe los reales*/
    char identificador[100]; /*recibe cadenas de caracteres*/
};

%token <numero> REAL/*token para real recibido de Lex*/
%token <identificador> ID/*token para identificador recibido de Lex*/
%token OPE /*token para operadores recibido de Lex*/
%token PARIZQ /*token para parentesis que abre recibido de Lex*/
%token PARDER /*token para parentesis que cierra recibido de Lex*/
%token CRLF /*token para salto de linea recibido de Lex*/

%start INICIO/*inicio de la gramatica*/
/*Seccion de reglas*/
%%
INICIO: GRAM GRAM | GRAM    {/*Inicio de la gramticas
                              Llama a gramatica para ver si avanza o no*/};
GRAM:   E CRLF              {printf("Cadena Aceptada\n");
                              /*Si la cadena la reconoce la gramatica y hay salto
                              de linea la cadena es aceptada y realiza un salto de linea*/ }
        | error CRLF        {yyerrok;
                              /*Si hay un error la gramatica no la reconece y avanza*/};
E:      T
        | E OPE T;
T:      ID
        | REAL
        | PARIZQ E PARDER;

%%
void yyerror(char *msg){
    /*imprime el error e imprime la linea en donde se encuentra*/
    printf("\tError Sintactico en linea %d\n",yylineno);
}
/*void main(){
    yyparse();
}*/
```

Conclusión

En este proyecto se repaso el funcionamiento del analizador sintáctico. A partir de dicho conocimiento se han identificado los tipos de analizadores sintácticos, tanto descendentes como ascendentes, nos hemos enfocado en desarrollar un analizador sintáctico descendente con ayuda de lex y yacc, investigando y analizado el funcionamiento de cada uno hemos logrado obtener un código eficiente que soluciona el problema propuesto.

Bibliografía

https://www.cartagena99.com/recursos/alumnos/apuntes/PDL_08_Tema%205_Analisis%20sintactico%20ascendente.pdf
https://www.infor.uva.es/~mluisa/talf/docs/aula/A3-A6.pdf
https://www.ecured.cu/Yacc
https://www.cartagena99.com/recursos/alumnos/apuntes/ININF2_M4_U3_T2.pdf