1a Guía Compiladores

Nombre:Lopez Perez Alberto Andrei Grupo: 5CM4

-Defina **compilador**

Un programa que traduce otro programa escrito en código fuente o un programa equivalente en lenguaje objeto

-Cuáles son las dos partes de la compilación

1.- Análisis

2.- Síntesis

-Describa las 6 fases de un compilador

Analizador léxico, Analizador sintáctico, Analizador semántico, Generador de código intermedio, Optimizador de código y Generador de código

-Cuáles son los 8 módulos de un compilador

Analizador léxico, Analizador sintáctico, Analizador semántico, Generador de código intermedio, Optimizador de código, Generador de código, Administrador de tabla de símbolos y Manejador de errores

Falso o verdadero (F/V)

|  |  |
| --- | --- |
| 0.-A los terminales se les llama asi porque no pueden ser sustituidos | ( V ) |
| 1.-Que una secuencia de caracteres concreta sea un token depende del lenguaje | ( V ) |
| 2.-Las cadenas que pertenecen al lenguaje generado por una gramática están hechas solo de terminales | ( V ) |
| 3.-El **análisis léxico** lee la cadena de entrada de derecha a izquierda | ( F ) |
| 4.-El **análisis léxico** construye el árbol de análisis sintáctico | ( **F** ) |
| 5.-La secuencia de caracteres que forma un componente léxico es el **lexema** del componente | ( **V** ) |
| 6.-La gramática S → **a**S | S**a** | **a** se puede analizar con un **análizador sintáctico predictivo descendente recursivo** | ( **F** ) |
| 7.-El tipo de **yylval** no es el mismo que el de los elementos en la pila de YACC | ( F ) |
| 8.-La única forma de indicar el tipo de los elementos en la pila de YACC es usando #define YYSTYPE | ( F ) |
| 9.-El **código intermedio** debe ser fácil de generar | ( **V** ) |
| 10.- Un **esquema de traducción** es una **GLC** + **reglas semánticas** | ( F ) |
| 11.- Árbol de análisis sintáctico con anotaciones es sinónimo de **árbol decorado** | ( V ) |
| 12-Análisis sintáctico **descendente** es donde la construcción del árbol de análisis sintáctico se inicia en las hojas y avanza hacia la raíz | ( **F** ) |

|  |  |
| --- | --- |
| 13-Análisis sintáctico **ascendente** es donde la construcción del árbol de análisis sintáctico se inicia en las hojas y avanza hacia la raíz | ( **V** ) |
| *14.-*yylex() llama a yyparse() | ( **F** ) |
| 15.-yyparse() llama a yylex() | ( **V** ) |
| *16.-*yylex() retorna el **tipo** de **token** | ( **V** ) |
| 17.-yylval almacena el **lexema** | ( V ) |
| 18-HOC1 es una **calculadora** | ( V ) |
| 19-Las **variables** en HOC son de tipo entero | ( **F** ) |
| 20.-La notación **posfija** es una notación matemática libre de paréntesis y en esta notación los operadores aparecen después de los operandos | ( V ) |
| 21.-La **raíz** del árbol de análisis sintáctico se etiqueta con el símbolo inicial | ( V ) |
| 22.- Las **hojas** del árbol de análisis sintáctico se etiquetan con no terminales | ( **V** ) |
| 23.-En la notación **infija** la **asociatividad** y la **precedencia** se usan para determinar en qué orden hay que realizar las operaciones para **evaluar** una expresión | ( V ) |

Para qué sirve el **Análisis Léxico**

a) Para generar el código en lenguaje b) Nos dice si una cadena pertenece al ( **c** ) objeto lenguaje generado por una gramática

c) Para dividir una cadena en tokens d) Los compiladores no lo necesitan nunca

El comprueba que el orden en que el **analizador léxico** le va entregando los tokens es válido.

1. analizador semántico b) analizador sintáctico c) optimizador d) generador de codigo ( **b** )

Es una gramática que tiene cuatro componentes:

* 1. Un conjunto de componentes léxicos.
  2. Un conjunto de no terminales.
  3. Un conjunto de producciones, en el que cada producción consta de un no terminal, llamado lado izquierdo de la producción, una flecha y una secuencia de componentes léxicos y no terminales, o ambos, llamado lado derecho de la producción.

4..La denominación de uno de los no terminales como símbolo inicial.

|  |  |  |
| --- | --- | --- |
| a) Gramática Asociativa por la izquierda | b) Gramática recursiva | ( **c** ) |
| c) Gramática libre de contexto (GLC) | d) Gramática ambigua |  |

Cuál de las sigs. opciones no es sinónimo de las otras

a) Componente léxico b) no terminal c) token d) Símbolo gramatical ( **d** )

Es una gramática donde en el lenguaje que genera existe una cadena que tiene más de un árbol de análisis sintáctico.

|  |  |
| --- | --- |
| a) Gramática recursiva por la izquierda b) Gramática recursiva | ( **d** ) |
| c) Gramática libre de contexto d) Gramática ambigua  Si Una gramática contiene una regla de producción de la forma A *→* A entonces es una |  |
| a) Gramática recursiva por la izquierda b) Gramática ambigua | ( **a** ) |

c) Gramática libre de contexto d) ninguna de las anteriores

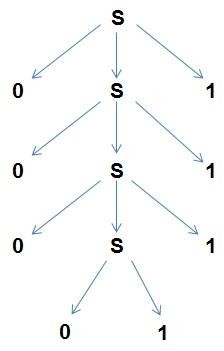
Considere la siguiente gramática

S*→* **0** S **1** | **01**

1. Mostrar una derivación de **00001111**

S  0S1  00S11  000S111  00001111

1. Dibuje el árbol de análisis sintáctico para la entrada **00001111**

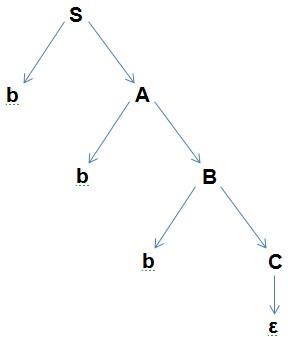


Considere la siguiente gramática S → **b**A

A → **b**B

B → **b**C C → **ε**

1. Mostrar una derivación de **bbb   
   S**  **bA**  **bbB**  **bbbC**  **bbbε**  **bbb**
2. Dibuje el árbol de análisis sintáctico para la entrada **bbb**



Considere la siguiente gramática

*S* → *A*

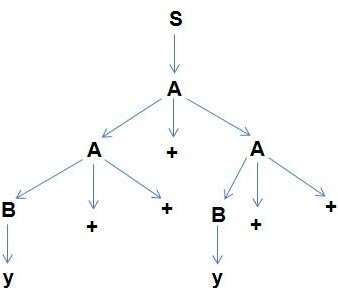
*A* → *A***+***A* | *B****++***

*B* → ***y***

1. Mostrar una derivación de **y + + + y + +**

S  A  A + A  B++ + A  B++ + B++  y++ + B++  y+++y++

1. Dibuje el árbol de análisis sintáctico para la entrada **y + + + y + +**



Considere la siguiente gramática

*l→ l* ***,*** *d | d*

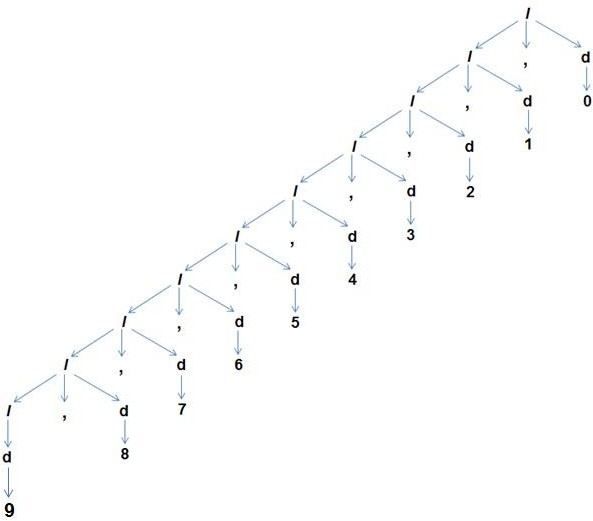
*d→* ***0*** *|* ***1*** *|* ***2*** *|* ***3*** *|* ***4*** *|* ***5*** *|* ***6*** *|* ***7*** *|****8*** *|* ***9***

a ) Mostrar una derivación de ***9,8,7,6,5,4,3,2,1,0***

*l*  *l, d*  *l, d, d*  *l, d, d, d*  *l, d, d, d, d*  *l, d, d, d, d, d*  *l, d, d, d, d, d, d*  *l, d, d, d, d, d, d, d*  *l, d, d, d, d, d,*

*d, d, d*  *l, d, d, d, d, d, d, d, d, d*  *d, d, d, d, d, d, d, d, d, d*  *9, 8, 7, 6, 5, 4, 3, 2, 1, 0*

b ) Dibuje el árbol de análisis sintáctico para la entrada ***9,8,7,6,5,4,3,2,1,0***



Dada la gramática

T= {**a**, **b**, +, -, \*, /, (,)}, N= {E, T, F} S= {E} P= { E->T | E+T | E-T

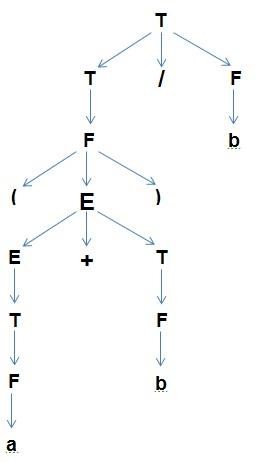
T->F | T\*F | T/F F-> **a** | **b** | (E) }

y la cadena **(a+b)/b**

1. Obtenga una derivación de dicha cadena

T  T/F  F/F  (E)/F  (E+T)/F  (T+T)/F  (F+T)/F  (F+F)/F  (a+F)/F  (a+b)/F  (a+b)/b

1. Dibuje el árbol de análisis sintáctico que corresponde a la cadena mencionada



# Análisis sintáctico predictivo descendente recursivo

Considere la siguiente gramática

S →**a** | **(** S **)**

Escriba el analizador sintáctico predictivo descendente recursivo

int preana //globalvoid parea(complex t

int parea(int t){ if(preanalisis == t

){

)

void main(){

preana

= sigcomplex(); if(preana == t) preanalisis == si

preana == sigcomplex();

}

gcomplex();

S(); else

else error(); void error(){

error(); puts(“ERROR”);

}

}

void S(){

}

} int sigcomplex(){if( preanalisis == '(' ){

return getchar();parea( '(' );

S();

void S(){ parea( ')' );

if( preana == '(' ){ }

else if (parea( '(' ); S(); parea( ')' ); preanalisis == 'a' )

} parea('a'); else if ( preanaelse == 'a' )

parea('a'); error();

}

} else error();

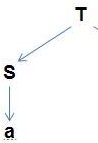
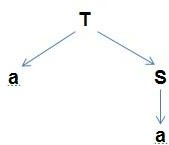
# Ambigüedad

Demostrar que la siguiente gramática es ambigua S → **a**S | S**a** | **a**

Usando la cadena **aa**

Es ambigua ya que la cadena “aa” produce más de 1 árbol de análisis sintáctico, y son los siguientes:

1) S  aS  aa



|  |  |
| --- | --- |
| 2) S Sa  aa  **S S** | Como produce más de un árbol de análisis sintáctico para la misma cadena, entonces la gramática es ambigua |

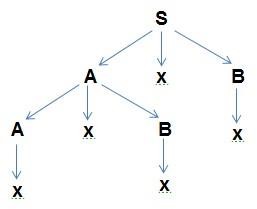
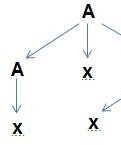
Demostrar que la siguiente gramática es ambigua A →A **x** B | **x**

B → **x** B | **x**

Usando la cadena **xxxxx**

De igual manera es ambigua ya que la cadena “xxxxx” produce más de 1 árbol de análisis sintáctico, y son los siguientes:

1. A  AxB  AxBxB  xxBxB  xxxxB  xxxxx
2. A  AxB  AxxB  AxxxB  xxxxB  xxxxx



|  |  |
| --- | --- |
| **A** | Como produce más de un árbol de análisis sintáctico para la misma cadena, entonces la gramática es ambigua |

Demostrar que la siguiente gramática es ambigua

S→ **a** S **b** S | **b** S **a** S | **ε**

Usando la cadena **abab**

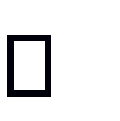
1) S  aSbS  abSaSbS  abεaεbε  abab

|  |  |  |
| --- | --- | --- |
| 2) S  aSbS  aSbaSbS  aεbaεbε  abab |  | Como produce más de un árbol de análisis sintáctico para la misma cadena, entonces la gramática es ambigua |

Verificar si las siguientes gramáticas son ambiguas

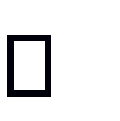
S → S **+** S | S **-** S | **a**

S → S S **+** | S S **-** | **a**

 Usando la cadena **a+a-a+a** obtenemos para la primer gramática

1. S  S+S  S+S+S  S+S-S+S  a+S-S+S  a+a-S+S  a+a-a+S  a+a-a+a
2. S  S-S  S+S-S+S  a+S-S+S  a+a-S+S  a+a-a+S  a+a-a+a

Como podemos generar la misma cadena de 2 formas distintas, **LA PRIMER GRAMÁTICA ES AMBIGUA**

 Usando la cadena **aa+a+a-a-** obtenemos para la segunda gramática

1. S  SS-  SS-S-  SS+S-S-  SS+S+S-S-  aS+S+S-S-  aa+S+S-S-  aa+a+S-S-  aa+a+a-S-  aa+a+a-a-

**LA SEGUNDA GRAMÁTICA NO ES AMBIGUA**, debido a que maneja la notación postfija

# Recursividad por la izquierda

Para eliminar la recursividad por la izquierda A → A**a** | **b**

se transforma en

A → **b** | **b**R R → **a**R | **ε**

Ahora considere las siguientes gramáticas

A→ **1** | A **0**

y

S→ **(** L **)** | **a**

L→ L **,** S | S

Elimine la recursividad por la izquierda de dichas gramáticas.

1) A  1R 2) S  ( L ) | a R  0R | ε L  S R

R  , S | ε

Escriba el analizador sintáctico predictivo descendente recursivo para dichas gramáticas

|  |  |
| --- | --- |
| **1 )** | void parea(complex t){ if(pre preanalisis == isgcom else error();  } void A(){ if( parea== '1'  ){ parea( '1' );  R();  } else error();  } void R(){ if(parea=='0'){ parea('0');  R();  } else error();  } |

|  |  |
| --- | --- |
| **2)** | void parea(complex t){ if(preanalisis == t) preanalisis == isgcomplex();  else error();  } void S(){ if( parea== '('  ){ parea( '(' ); L();  parea( ')' );  }  else if (parea == 'a') parea('a'); else error();  } void L(){  S();  R();  } void R(){ if(parea==','){ parea(',');  S();  } else error();  } |

Escriba la sección de reglas de la especificación de YACC para dichas gramáticas

//Gramatica 1)

%%

A: '1' R

;

R: /\*Epsilon (Na '0' R

;

%%

//Gramatica 2)

%%

S: '(' L ')'

| a

;

L: S R

;

R: /\*Epsilon (Nada)\*/ ',' S

;

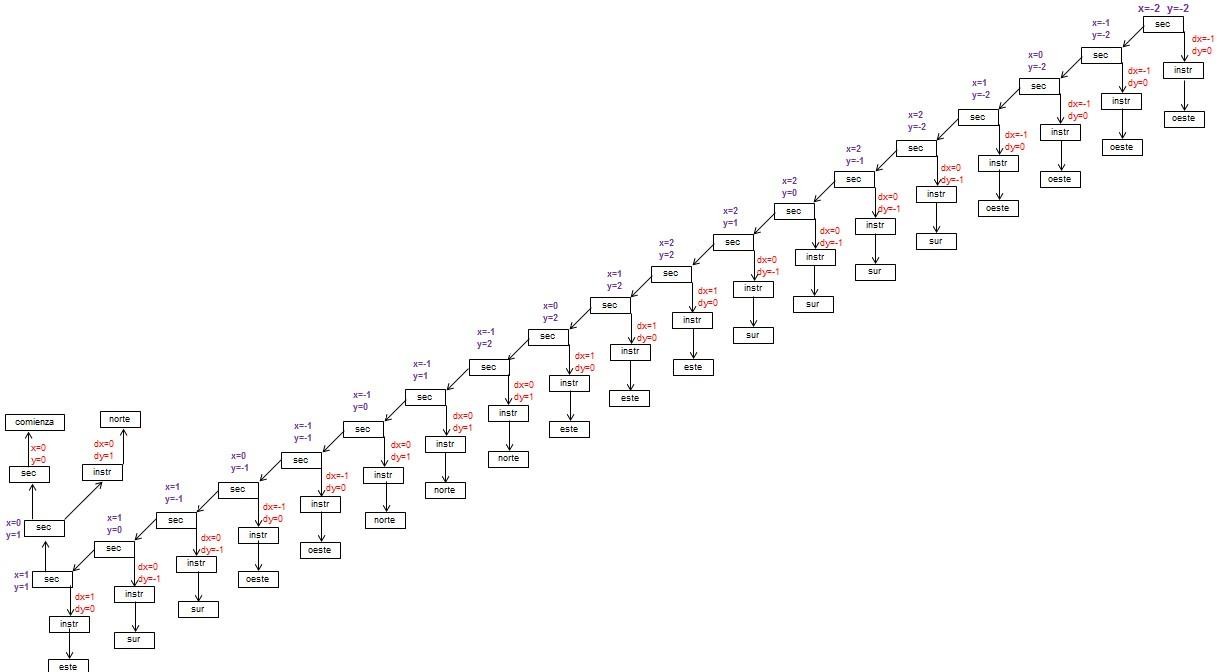
%%

# Definiciones dirigidas por la sintaxis

|  |  |
| --- | --- |
| PRODUCCIÓN | REGLA SEMÁNTICA |
| *sec →* **comienza** | *sec.x = 0*  *sec.y = 0* |
| *sec → sec1 instr* | *sec.x = sec1.x + instr.dx sec. y = sec1.y + instr.dy* |
| *instr →* **este** | *instr.dx = 1 instr.dy*  *= 0* |

|  |  |
| --- | --- |
| *instr →* **norte** | *instr.dx = 0 instr.dy*  *= 1* |
| *instr →* **oeste** | *instr.dx = -1 instr.dy*  *= 0* |
| instr → **sur** | *instr.dx = 0 instr.dy*  *= -1* |

Dibuje el árbol de análisis sintáctico con anotaciones para la sig cadena c n e ss oo nnn eee ssss oooo



Escribir la sección de reglas de la especificación de yacc para calcular la posición final del robot.

%{

struct cord{

int x, y, dx, dy;

} ;

Typedef struct cord cordenada; #define struct cord cordenada #define YYSTYPE struct cord

%}

%token comienza este oeste norte sur

%%

sec: comienza {$$.x=0;$$.y=0;}

| sec instr {$$.x=$1.x + $2.dx;$$.y=$1.y + $2.dy;}

;

instr: este{$1.dx=1; $1.dy=0;}

| oeste{$1.dx=-1; $1.dy=0;}

| norte{$1.dx=0; $1.dy=1;}

| sur{$1.dx=0; $1.dy=1;}

;

%%

Escriba una definición dirigida por la sintaxis para evaluar expresiones booleanas.

|  |  |
| --- | --- |
| **Producción** | **Regla semántica** |
| expr  expr1 '|' expr2 | expr.t = expr1.t || expr2.t |
| expr  expr1 '&' expr2 | expr.t = expr1.t && expr2.t |
| expr  '!' expr1 | expr.t = !expr1.t |

# Esquemas de traducción

Escriba un esquema de traducción para convertir una expresión en:

|  |  |
| --- | --- |
| 1. infijo a postfijo | 2. postfijo a infijo |
| 3. infijo a prefijo | 4. prefijo a infijo |

Escriba un esquema de traducción para evaluar expresiones booleanas

* 1. **Infijo a Postfijo** expr → expr + termino

{printf(‘+’)} expr → expr - termino {printf(‘- ’)}

expr → termino

termino → 0 {printf(‘0’)} termino

→ 1 {printf(‘1’)}

…

termino → 9 {printf(‘9’)}

* 1. **Postfijo a Infijo** expr → + expr termino +

{printf(‘+’)} expr → - expr termino - {printf(‘- ’)} expr → termino termino → 0 {printf(‘0’)} termino → 1 {printf(‘1’)}

…

termino → 9 {printf(‘9’)}

* 1. **Infijo a Prefijo** expr → expr termino +

{printf(‘+’, expr, termino)} expr → expr termino – {printf(‘-’, expr, termino)} expr → termino

termino → 0 {printf(‘0’)} termino

→ 1 {printf(‘1’)}

…

termino → 9 {printf(‘9’)}

* 1. **Prefijo a Infijo** expr → + expr termino

{printf(expr, ‘+’, termino)} expr → - expr termino {printf(expr, ’-’, termino)} expr → termino

termino → 0 {printf(‘0’)} termino

→ 1 {printf(‘1’)}

…

termino → 9 {printf(‘9’)}

Para cada esquema de traducción de arriba escriba la sección de reglas de la especificación de YACC

|  |  |  |
| --- | --- | --- |
| **2. Infijo a Postfijo** expr: expr '+' termino  | expr '-' termino  | termino  ;  termino: '0'  | '1'  …  | '9'  ; |  | **1. Postfijo a Infijo** expr: '+' expr termino '+'  | '-' expr termino '-'  | termino  ;  termino: '0'  | '1'  …  | '9'  ; |

|  |  |  |
| --- | --- | --- |
| **4. Infijo a Prefijo** expr: expr termino '+'  | expr termino '-'  | termino  ;  termino: '0'  | '1'  …  | '9'  ; |  | **3. Prefijo a Infijo** expr: '+' expr termino  | '-' expr termino  | termino  ;  termino: '0'  | '1'  …  | '9'  ; |

# Escritura de Gramáticas

Escribir una gramática que genere todas las cadenas de longitud 4 formadas con los símbolos del alfabeto {a,b,c}

T = {a, b, c} /\*Terminales\*/

N = {A, S} /\*No terminales\*/ S = {S} /\*Símbolo Inicial\*/

P = {S → AAAA /\*Producciónes\*/ A → a | b | c}

# Escribir una gramática que sirva para generar las siguientes cadenas

|  |  |  |
| --- | --- | --- |
| **Especie perro Edad 1**  **Sexo macho Tamaño grande**  **Colores negro , blanco Soy rápido , activo, alegre**  **Aficiones correr, comer** | **Especie gato Especie perro**  **Edad 2 Edad 2**  **Sexo macho Sexo hembra**  **Tamaño mediano Tamaño pequeño Colores negro , blanco , café Colores canela , gris**  **Soy tranquilo , sociable Soy fuerte , alegre, activo.**  **Aficiones dormir, Aficiones aullar parrandear, comer** | **Especie gato Edad 2**  **Sexo macho Tamaño grande Colores blanco**  **Soy listo , obediente**  **Aficiones jugar, haraganear** |

S  especie + edad + sexo + tamaño + colores + soy + aficiones Especies  perro | gato

Edad  1 | 2

Sexo  macho | hembra

Tamaño  grande | mediano | pequeño

Colores  colores, colores | colores | negro | blanco | café | canela | gris

Soy  soy, soy | soy | rápido | activo | alegre | tranquilo | sociable | fuerte | listo | obediente

Aficiones  aficiones, aficiones | aficiones | correr | comer | dormir | parrandear | aullar | jugar | haraganear

12.-Escribir una gramática que sirva para generar las siguientes cadenas

Etiquetado Nerd Nivel Junior Sexo Hombre

Lenguajes Java , C , Logo Aficiones programar,

Etiquetado Geek Nivel Senior Sexo Mujer

Lenguajes Pascal , Prolog , SQL

Etiquetado Nerd Etiquetado Freak

Nivel Junior Nivel Senior

Sexo Mujer Sexo Hombre

Lenguajes PHP , Perl, Java Lenguajes Ensamblador, C Aficiones hackear, googlear, Aficiones gotcha, dormir,

videogames, comics, hackear, googlear

Aficiones chatear,gotcha, dormir chatear, comics

videogames, programar

S  etiquetado + nivel + sexo +lenguajes + aficiones Etiquetado  nerd | geek | freak

Nivel  junior | senior Sexo  hombre | mujer

Lenguajes  lenguajes, lenguajes | lenguajes | java | c | logo | pascal | prolog | pearl | php | ensamblador

Aficiones  aficiones, aficiones | aficiones | programar | videogames | comics | hackear | googlear | chatear | dormir | gotcha

# YACC

.-Para qué sirve $$

Para guardar el valor que se le asigna a toda la producción

.-Dentro de una acción gramatical $n se refiere al **n-esimo símbolo gramatical del lado derecho de la**

|  |  |
| --- | --- |
| **producción**  1.-Los %% se usan para indicar |  |
| a)inicio de la sección de declaraciones | b)inicio de la sección de reglas ( **b** ) |
| c)precedencia de los operadores  2.-%token sirve para indicar | d)fin del código de soporte |
| a)inicio de la sección de declaraciones | b)los no terminales de la gramática ( **d** ) |
| c)precedencia de los operadores | d)los terminales de la gramática |

3.-Como le indica el analizador léxico (yylex) al analizador sintáctico (yyparse) que ya no hay más tokens en la entrada

|  |  |  |  |
| --- | --- | --- | --- |
| a) retornando cero  c) almacenando -1 en yylval  4.-Una acción gramatical debe ir entre | b) retornando -1  d) almacenando 0 en yylval | ( **a** | ) |
| a) comillas b) paréntesis c) corchetes d) llaves 5.-Considere la producción  S : S ' a' S 'b'  $4 a cuál de los miembros del lado derecho de la producción se refiere? | | ( **d** | ) |
| a)la 'a'  c)la segunda S  Si el código de yylex es el siguiente int yylex() { return getchar(); } de cuantos caracteres son los tokens | b)la 1er S d)la 'b' | ( **d** | ) |
| a) 0 | b) 1 c) 2 d) la cantidad de caracteres del token varia | ( **b** | ) |
| Considere la siguiente gramática (los terminales se indican en negritas) | |  | |

L-> L**,** D | D D-> **0** | **1**

Escriba la sección de reglas de la especificación de yacc para dicha gramática

%%

L: L ',' D

| D

;

D: 0

| 1

;

%%

Escriba la especificación de yacc para la gramática

S → U | V

U → T**a**U | T**a**T V → T**b**V | T**b**T

T → **a**T**b**T | **b**T**a**T | **ε**

Especificaciones:

%%

S: U

| V

;

U: T 'a' U

| T 'a' T

;

V: T 'b' V

| T 'b' T

;

T: /\*Nada (épsilon)\*/

| 'a' T 'b' T

| 'b' T 'a' T

%%

Escriba las acciones gramaticales para que imprima el número de b's en la cadena de entrada

%{

int cantidad;

/\*escriba el tipo de los elementos en la pila de yacc \*/ #define YYSTYPE int

%}

%%

S: ’(’ B ’)’ {$$ = $2;}

;

B: ’(’ B ’)’ {$$ = $2;}

| D {$$=$1; }

;

D:

| ’b’ D {$$.cantidad++; $$=$2;}

;

%%

Considere la siguiente gramática (los terminales se indican en negritas) lista->lista **,** figura | figura figura-> triangulo | cuadrilatero triangulo-

> **lado lado lado** cuadrilatero-> **lado lado lado lado**

Escriba la sección de reglas de la especificación de yacc para dicha gramática y las acciones semánticas respectivas para que se imprima si un triángulo es equilátero y si un cuadrilátero es un cuadrado

%%

lista: lista ',' figura

| figura

;

figura: triangulo

| cuadrilatero

;

triangulo: lado lado lado {if($1==$2 && $2==$3) printf("Triangulo equilatero");}

;

cuadrilatero: lado lado lado lado {if($1==$2 && $2==$3 && $3==$4) printf("Cuadrilatero");}

;

%%