

**Analizador Sintáctico descendente con retroceso**

Universidad Tecnológica Nacional - Facultad regional Delta

Materia: Sintaxis y semántica de los lenguajes

Alumno: KUCHARUK, Manuel

Docentes: SANTOS, Juan; MIRANDA, Hernán

Fecha: 26/10/2015

Índice

[Gramática y estructuras utilizadas 3](#_Toc433524064)

[Código del programa 4](#_Toc433524065)

[Observaciones y comentarios 5](#_Toc433524066)

[Ejemplo de ejecución 6](#_Toc433524067)

[Conclusión 7](#_Toc433524068)

# Gramática y estructuras utilizadas

Usé la gramática completa de la consigna del TP, pero modificada para evitar reglas borradoras y recursividad izquierda.

Como el analizador sintáctico usa antes un analizador léxico, algunos símbolos no terminales pasan a ser terminales en la nueva gramática; por ejemplo NombreVariable, NombreFuncion, Operador, etc.

Símbolos No Terminales = {P, DV, DF, LP, LPCont, LC, CAsignacion, CPara, CSiEntonces, CLLamadoFuncion, LLPar, LLParCont, ExpEntera, ExpEntera2, Termino, Termino2, Factor, ExpLogica, ExpLogica2, OperandoLogico, OperandoLogico2, FactorLogico, Comparacion}

Producciones =

**P** -> DV DF LC | DF LC | DV LC | DV DF | DV | DF | LC

**DV**-> Tipo NombreVariable; DV | Tipo NombreVariable;

**DF**-> Tipo NombreFuncion LP {LC; return NombreVariable;} DF

| Tipo NombreFuncion LP {LC; return NombreVariable;}

**LP**-> (LPCont) | ()

**LPCont**-> Tipo NombreVariable, LPCont | Tipo NombreVariable

**LC**-> CAsignacion LC | CPara LC | CSiEntonces LC | CLLamadoFuncion LC

| CAsignacion | CPara | CSiEntonces | CLLamadoFuncion

**CAsignacion**-> NombreVariable = ExpEntera;

**CPara**-> Para NombreVariable desde ExpEntera hasta ExpEntera {LC}

| Para NombreVariable desde ExpEntera hasta ExpEntera {}

**CSiEntonces**-> Si ExpLogica entonces {LC} | Si ExpLogica entonces {}

**CLLamadoFuncion**-> NombreFuncion LLPar;

**LLPar**-> (LLParCont) | ()

**LLParCont**-> NombreVariable, LLParCont | NombreVariable

**ExpEntera**-> Termino ExpEntera2 | Termino

**ExpEntera2**-> + Termino ExpEntera2 | - Termino ExpEntera2 | + Termino | - Termino

**Termino**-> Factor Termino2 | Factor

**Termino2**-> \* Factor Termino2 | \* Factor

**Factor**-> (ExpEntera) | ConstEntera | NombreVariable

**ExpLogica**-> OperandoLogico ExpLogica2 | OperandoLogico

**ExpLogica2**-> O OperandoLogico ExpLogica2 | O OperandoLogico

**OperandoLogico**-> FactorLogico OperandoLogico2 | No FactorLogico OperandoLogico2

| FactorLogico | NO FactorLogico

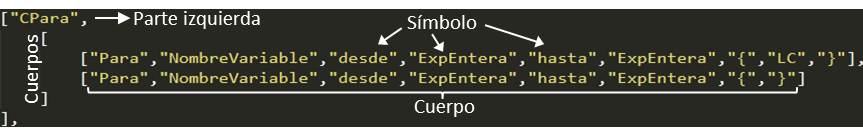
**OperandoLogico2**-> Y FactorLogico OperandoLogico2 | Y FactorLogico

**FactorLogico**-> (ExpLogica) | ConstLogica | Comparacion

**Comparacion**-> ExpEntera Operador ExpEntera

Las gramáticas están definidas en el archivo producciones.js en una variable tipo array llamada producciones.

Cada elemento del array de producciones también es un array que se compone de la siguiente manera:



El primer elemento es la parte izquierda de la producción (un símbolo no terminal), el segundo elemento es otro array, que contiene todas las producciones (cuerpos) asociadas a la parte izquierda.

Cada cuerpo es un nuevo array compuesto por símbolos.

En la imagen, se muestra la representación de la producción:

CPara -> Para NombreVariable desde ExpEntera hasta ExpEntera { LC }

Sabiendo que LC puede ser λ, se crea también la producción:

CPara -> Para NombreVariable desde ExpEntera hasta ExpEntera { }

# Código del programa

De index.js:

var pathLexer = path.join(\_\_dirname,"../bin","lexico.exe");

lexer.analizar(pathLexer,req.body.entrada,function (resLexer){

parser.analizar(resLexer+"$",function (resParser){

res.render("index",{

entrada: req.body.entrada,

salida: resLexer,

resultadoParser: formatearResultado(resParser)

}

}

}

lexer.analizar(pathLexer,req.body.entrada,function (resLexer){:

El programa llama al ejecutable lexico.exe, que es el analizador del TP1, analizando el contenido del textbox entrada y poniendo el resultado en la variable resLexer.

parser.analizar(resLexer+"$",function (resParser){:

Se ejecuta el analizador sintáctico con la cadena resLexer concatenada con el símbolo de fin de cadena $. El resultado se almacena en resParser (una variable que simplemente vale 0 si la cadena no fue aceptada y 1 si la cadena sí fue aceptada).

res.render("index",{

entrada: req.body.entrada,

salida: resLexer,

resultadoParser: formatearResultado(resParser)

Muestra la entrada del usuario, el resultado del análisis léxico y si la cadena fue aceptada o no por el analizador sintáctico.

De parser.js:

La implementación del algoritmo es básicamente la misma que el pseudocódigo provisto en los apuntes. Excepto por las siguientes modificaciones:

Cada función devuelve la posición a la que se está apuntando en la cadena de entrada, en vez de ser una variable global. De esa forma se puede volver hacia atrás en la función procesar en caso de error.

La función PNi, en vez ser una función por cada no terminal, es una única función que recibe un símbolo no terminal y busca las producciones (cuerpos) asociadas a ese símbolo.

# Observaciones y comentarios

Hice en analizador usando el lenguaje Javascript, que permite usar datos directamente desde un navegador de Internet y sin preocuparse por la interfaz gráfica. Además, para hacer la llamada al analizador léxico (archivo ejecutable de Windows), use Node,js, que permite ejecutar Javascript en un servidor web.

Saqué el símbolo NombreVariable de la siguiente producción:

FactorLogico -> (ExpLogica) | ConstLogica | NombreVariable | Comparacion

Si estuviera presente, una expresión del tipo Si a==b entonces { } daría error, ya que la expresión a==b solamente haría un match correcto con a como NombreVariable, pero no llegaría a reconocer la comparación completa.

La producción DF -> Tipo NombreFuncion LP { LC ; return NombreVariable ; }DF obliga a que haya un ; extra antes de return, independientemente de que todas las producciones de LC terminen con ; o }.

# Ejemplo de ejecución

**Entrada:**

entero suma;

entero n;

real tiempo;

entero Sumar(entero a, entero b){

suma = a+b;

;

return suma;

}

tiempo=0;

Para i desde 0 hasta 150{

Si (tiempo>=0 Y tiempo<20) O tiempo==2 entonces{

suma = suma+1;

tiempo = tiempo + 1;

n = (tiempo + i)\*suma-n;

Mostrar(i,suma,tiempo,n);

}

}

**Salida** (saltos de línea agregados):

<Tipo><NombreVariable><;>

<Tipo><NombreVariable><;>

<Tipo><NombreVariable><;>

<Tipo><NombreFuncion><(><Tipo><NombreVariable><,><Tipo><NombreVariable><)><{>

<NombreVariable><=><NombreVariable><+><NombreVariable><;>

<;>

<return><NombreVariable><;>

<}>

<NombreVariable><=><ConstEntera><;>

<Para><NombreVariable><desde><ConstEntera><hasta><ConstEntera><{>

<Si>

<(><NombreVariable><Operador><ConstEntera><Y>

<NombreVariable><Operador><ConstEntera><)>

<O><NombreVariable><Operador><ConstEntera><entonces><{>

<NombreVariable><=><NombreVariable><+><ConstEntera><;>

<NombreVariable><=><NombreVariable><+><ConstEntera><;>

<NombreVariable><=><(><NombreVariable><+><NombreVariable><)><\*>

<NombreVariable><-><NombreVariable><;>

<NombreFuncion><(>

<NombreVariable><,>

<NombreVariable><,>

<NombreVariable><,>

<NombreVariable><)><;>

<}>

<}>

**Cadena aceptada**

# Conclusión

El parser con backtracking permite de una manera no muy complicada hacer el análisis sintáctico de una cadena. Sirve para gramáticas sin reglas borradoras ni recursividad izquierda, lo cual hace que este tipo de parser sea más abarcativo que un parser descendente predictivo. Tiene el inconveniente que solamente acepta o rechaza la cadena provista, no permite producir el árbol de derivación.

La implementación del pseudocódigo en Javascript es directa salvo por las excepciones ya comentadas. Durante la realización del TP se nota la importancia de programar correctamente el analizador léxico (TP1), porque se tiene que “confiar” en la salida de este.