Compiladores

- Apuntes -

Grupo 3CM7

Vargas Romero Erick Efraín
Prof. Tecla Parra Roberto

Instituto Politécnico Nacional
Escuela Superior de Cómputo
Juan de Dios Bátiz, nueva industrial Vallejo
07738 ciudad de México

Este documento ha sido escrito utilizando la herramienta sharelatex con el fin de recopilar toda la información posible que se ha brindado en la clase de "Compiladores"

Contents

Chapter 1

Primer parcial

1.1 Introducción

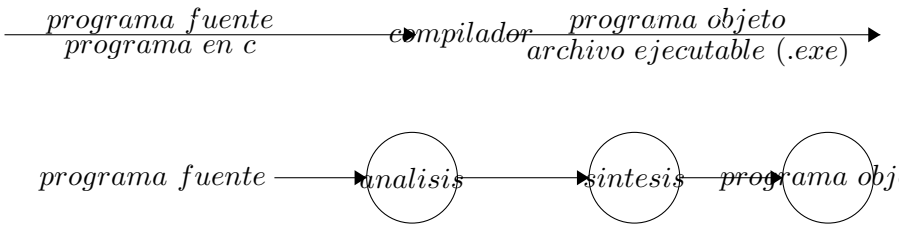
Debemos retomar un poco sobre lo que ocurría en la revolución industrial; Hay 3 factores importantes o causas importantes para la revolución industrial.

1. Motor de vapor
2. Telar
3. Máquinas herramienta (Torno)

Recordemos que también tenemos dos clasificaciones importantes en el software

- Base (sistema)
- Aplicación
    1. Ofimática
    2. Navegador
    3. Procesamiento digital
    4. videojuegos

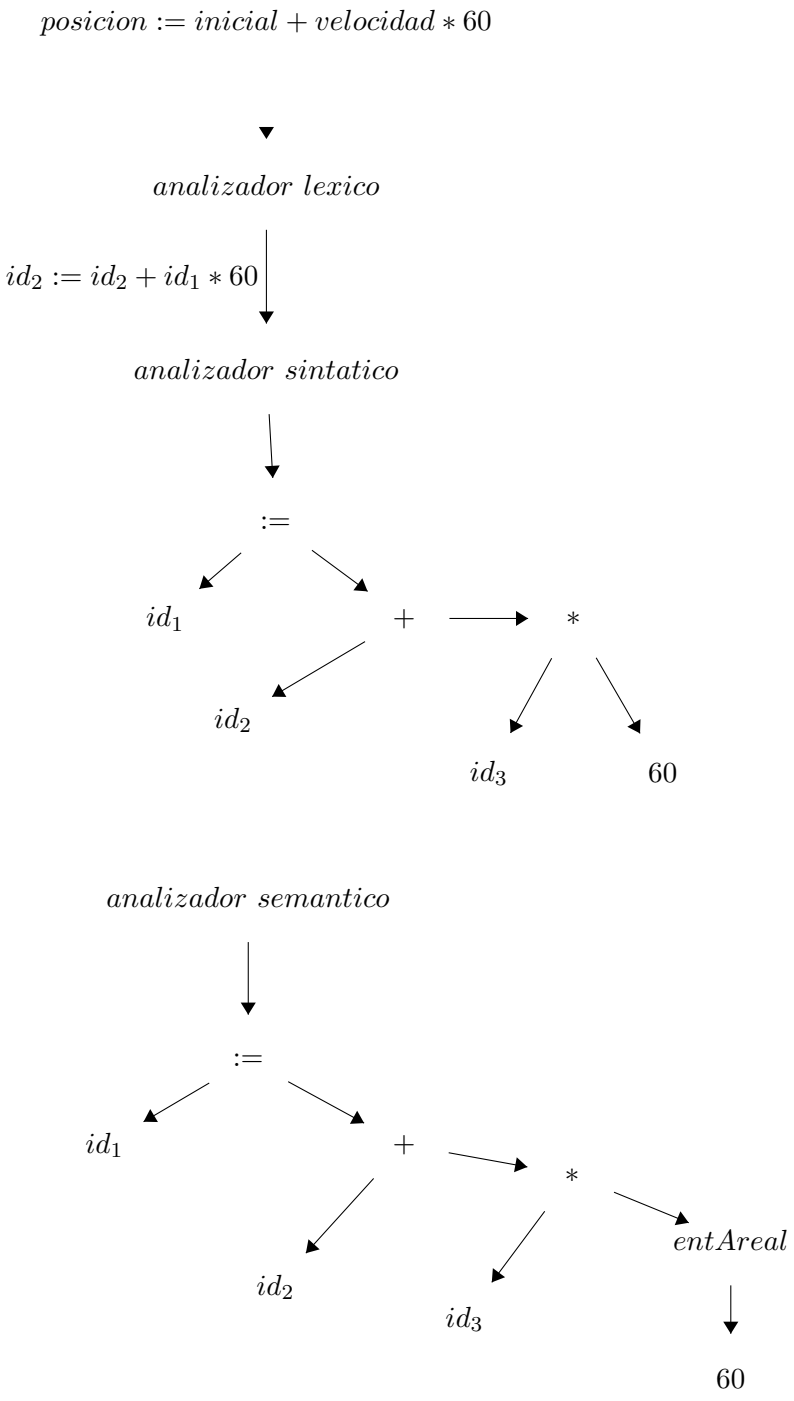
Un lenguaje de programación es una notación.
Un compilador es un programa que toma un lenguaje escrito en lenguaje fuente y lo convierte en su equivalente en lenguaje objeto.



Un interprete también ejecuta las instrucciones del programa.

¿Qué es el análsis? Separar algo en partes más pequeñas para comprender su funcionamiento

¿Qué es la síntesis? Contrario a el análisis, es el unir las partes más pequeñas



Analizador léxico Divide una cadena en tokens (Se realiza un análisis lineal) o componentes léxicos

Analizador sintáctico En este módulo se construye el árbol sintáctico, además se revisa que los tokens tengan el órden correcto. También verifica que la cadena de entrada pertenezca al lenguaje generado por la gramática.

Un componente lexico que tiene significado colectivamente

generador de codigo intermedio

$$\begin{aligned} tmp_1 &:= entAreal(60) \\ tmp_2 &:= id_3 * tmp_1 \\ tmp_3 &:= id_2 + tmp_2 \\ id_2 &:= tmp_3 \end{aligned}$$

optimizador de codigo

$$\begin{aligned} tmp_1 &:= id_1 * 60.0 \\ id_1 &:= id_2 + tmp_1 \end{aligned}$$

Generador de codigo

Codigo de 3 *direcciones*

Optimizador de código Reduce el número de instrucciones

Generador de código Genera código objeto

Código de 3 direcciones En este existen 3 operandos y 2 operadores

1.2 Gramática libre de contexto (GLC)

Tiene cuatro componentes

1. Un conjunto de componentes léxicos denominados símbolos terminales
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 terminales y no terminales, o ambos llamado lado derecho de la producción.
4. La denominación de uno de los no terminales como símbolo inicial

Una gramática es una cuádrupla que contiene componentes léxicos terminales

Token = componente léxico = terminal

Un componente léxico es un conjunto de caracteres que tienen significado colectivo

Producción Algo que tiene lado izquierdo y lado derecho

Example 1.1 (
Una producción
$P = \{lista \rightarrow lista + digito,$
$lista \rightarrow lista - digito,$
$lista \rightarrow digito,$
$digito \rightarrow 0|1|2|3|4|5|6|7|8|9\}$

Podemos decir entonces que G(T, N, P, S)
dónde

- T son los símbolos terminales
- N son los no terminales
- P las producciones
- S el símbolo inicial

Del ejemplo anterior nótese que
$T = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, +, -\}$
$N = \{lista, digito\}$
$S = \{lista\}$

Por tanto tenemos la siguiente estructura

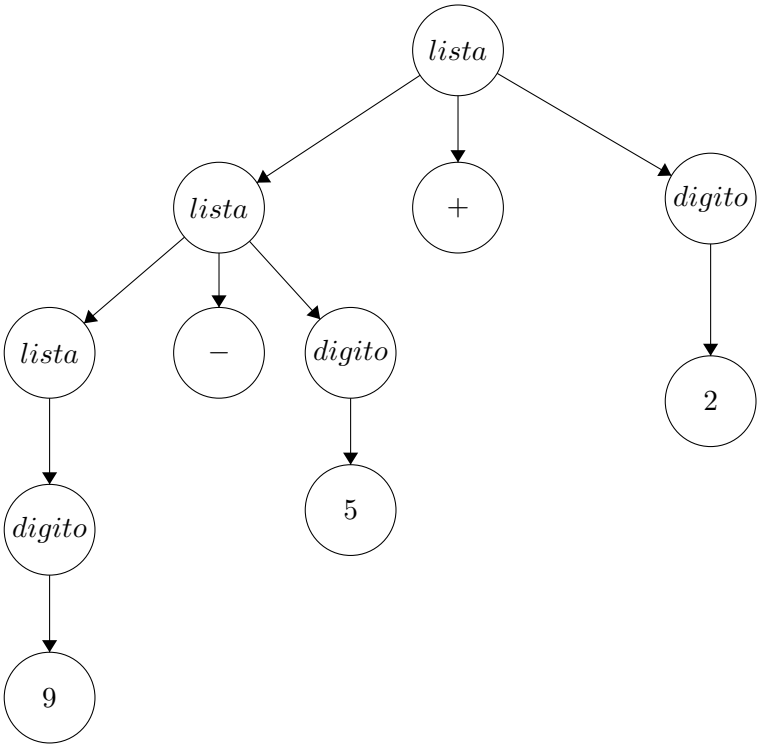
Example 1.2 (l)
ado izquierdo $\rightarrow$ lado derecho

¿Qué es una producción? En palabras simples una producción es una regla de sustitución.

Example 1.3 ()
9 - 5 + 2

Se representaría como
lista → lista + digito
→ lista - digito + digito
→ digito - digito + digito
→ 9 - digito + digito
→ 9 - 5 + digito
→ 9 - 5 + 2

El A.A.S resultante es



Un terminal es algo que no se puede sustituir, los terminales aparecen del lado derecho de la producción

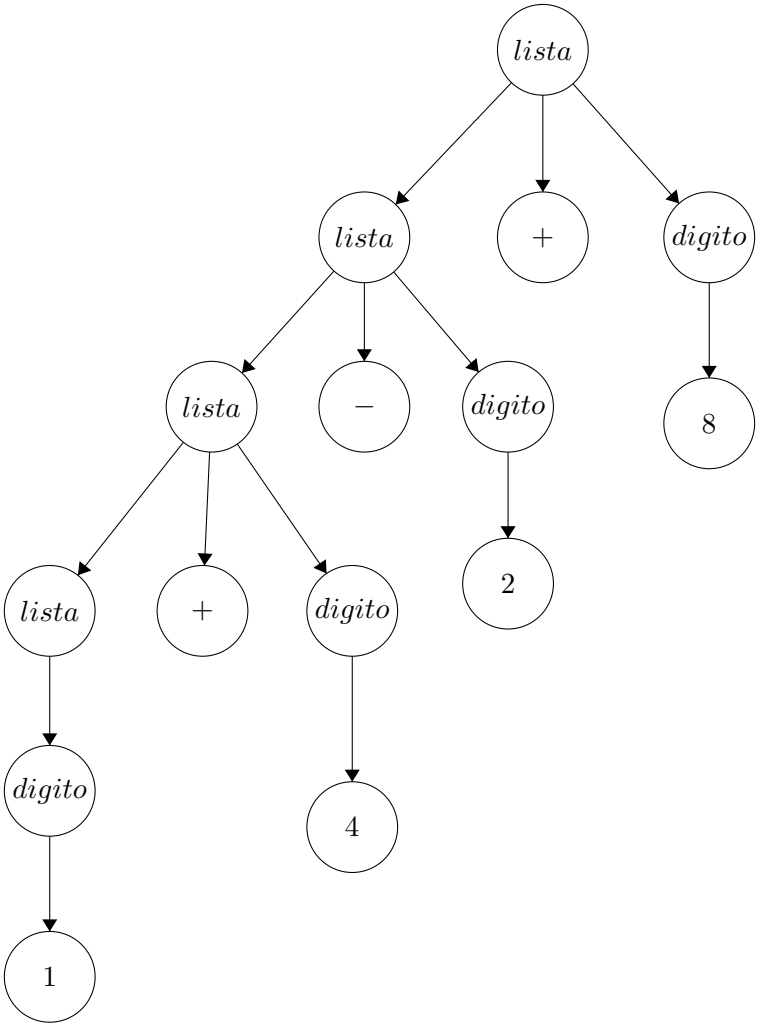
Símbolos terminales Un símbolo no terminal debe aparecer al menos una vez del lado izquierdo de la producción.

Example 1.4 ()
1 + 4 - 2 + 8

lista → lista + digito
→ lista - digito + digito
→ lista + digito - digito + digito
→ digito + digito - digito + digito

→ 1 + digito - digito + digito
→ 1 + 4 - digito + digito
→ 1 + 4 - 2 + digito
→ 1 + 4 - 2 + 8

El A.A.S resultante es



1.3 Árboles de análisis sintáctico (A.A.S)

UN A.A.S indica gráfica como del símbolo inicial de una gramática deriva na cadena del lenguaje.
EL A.A.S tiene las siguientes propiedades:

1. La raíz esta etiquetada con el símbolo inicial
2. Cada hoja esta etiquetada con un token o con ϵ
3. Cada nodo interior está etiquetado con un no terminal
4. Si A es el no terminal que etiqueta a algún nodo interior y $A \to X_1, X_2, ..., X_n$ son las etiquetas de los hijos de ese nodo de izquierda a derecha, entonces $A \to X_1, X_2, ..., X_n$

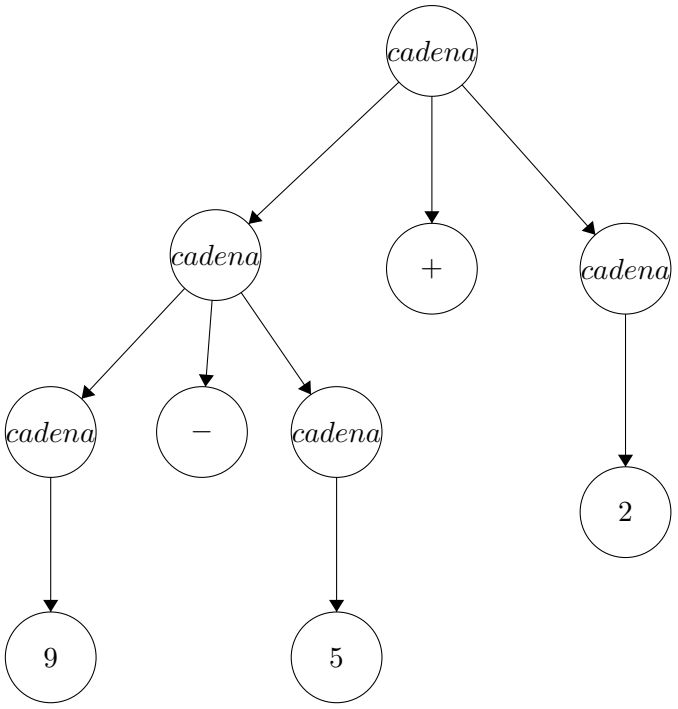
es no producido. Aquí $X_1, X_2, ..., X_n$ representan un símbolo gramatical como caso especial si $A \to \epsilon$ entonces un nodo etiquetado con A tiene un solo hijo etiquetado con ϵ

Ambigüedad Una gramática es ambigua si para una cadena del lenguaje tiene dos árboles de análisis sintáctico diferentes

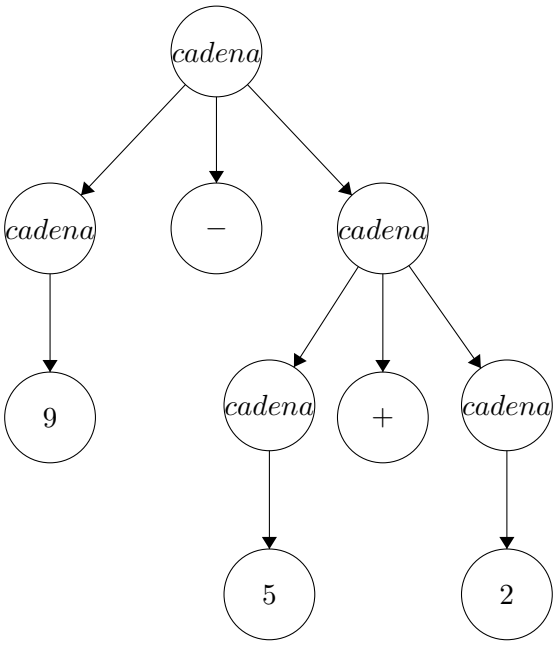
Example 1.5 ()
Consideremos lo siguiente:

cadena $\to$ cadena + cadena | cadena - cadena |0|1|2|3|4|5|6|7|8|9
$T = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, +, -\}$
$N = \{cadena\}$
$S = \{cadena\}$

Con lo cual obtenemos los siguientes A.A.S



El árbol anterior nótese que está más cargado a la izquierda



El árbol anterior nótese que está más cargado a la derecha

.

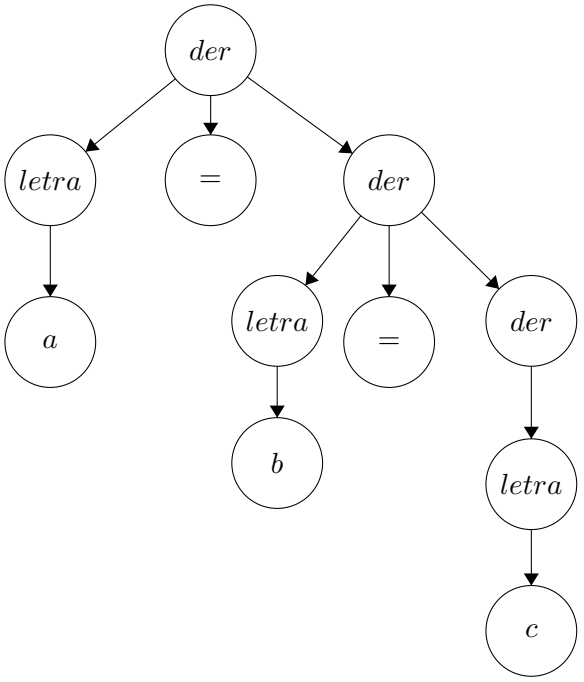
1.4 Asociatividad de operadores

der → letra = der | letra

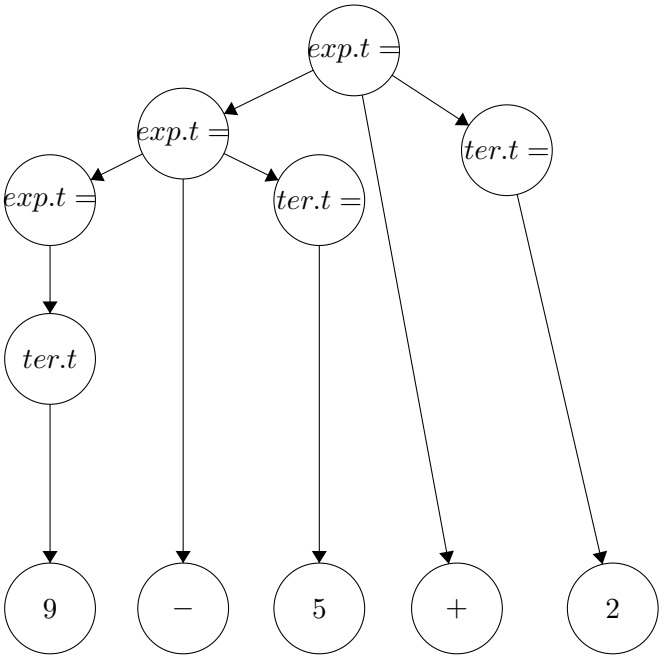
letra → a | b | c | ... | z

$T = \{a, ..., 7, =\}, N = \{letra, der\}, S = \{der\}$

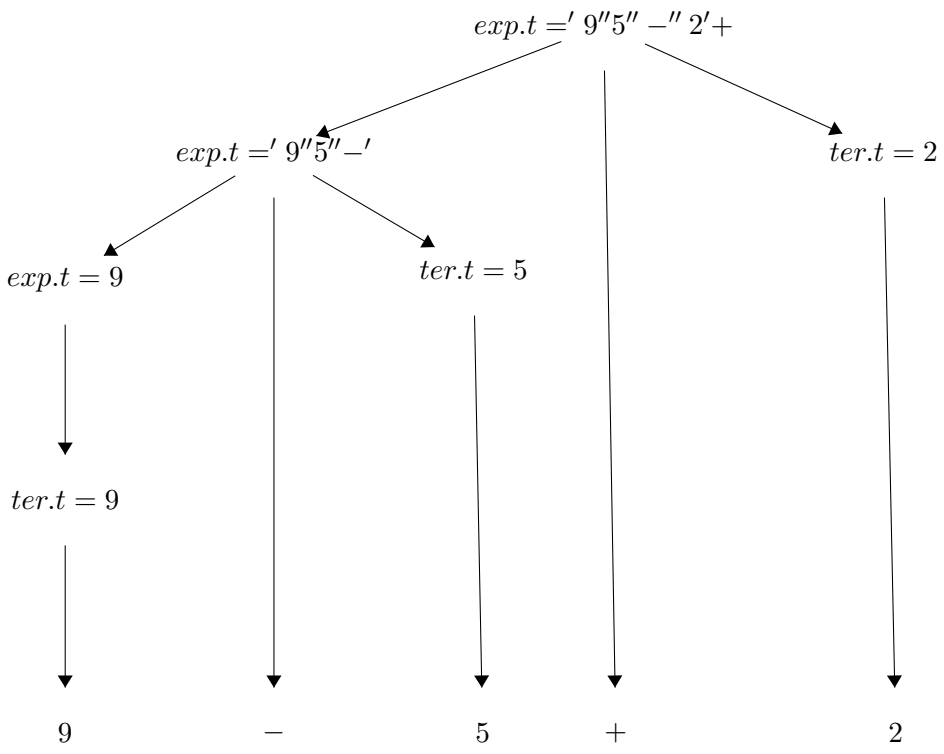
$a = b = c$
Por propiedades del operador equivalencia
$(a = (a + c))$



9 - 5 + 2 equiv. a



Resulta en



Podemos utilizar la asociatividad porque los operadores tienen la misma precedencia
$T = \{0, 1, ..., 9, +, -\} N = \{exp, ter\} S = \{exp\}$
asociatividad: Podemos utilizar la asociatividad cuando la precedencia es la misma. La utilizamos para evaluar y dar un órden de evaluación.

1. izq: suma, resta, multiplicación, división. Los operandos se agrupan de izquierda a derecha.
2. der: potencia, asignación. Es decir agrupamos de derecha a izquierda.
3. no asociativa:

1.4.1 Precedencia de operadores

exp → exp + ter | expo - ter | ter

ter → ter * fac | ter / fac | fac

fac → digito | (exp)

9 + 5 * 2 equiv a
No podemos utilizar solo la asociatividad porque tenemos un operador de multiplicación
Los paréntesis sirven para cambiar el órden de operación. Resolvemos primero lo que esté dentro de los paréntesis.
Cuando queremos cambiar el órden de evaluación utilizamos el paréntesis

1.5 Traducción dirigida por la sintaxis

1.5.1 Notación postfija

La notación postfija de una expresión E se puede definir de manera individual como sigue:

1. Si E es una variable o una CTE entonces la notación postfija de E es también E.
2. Si E es una expresión de la forma E_1 operador E_2 entonces la notación postfija de E es $E_1' E_2'$ operador donde E_1' y E_2' son las notaciones postfijas de E_1 y E_2 respectivamente.
3. Si E es una expresión de la forma (E_1) entonces la notación postfija de E_1 es también la notación postfija de E. (No hay paréntesis)

Example 1.6 (9)
-5+2 en postfijo 95-2+
INFIJO
(7 + 9) en postfijo 79+

1.5.2 Definiciones dirigidas por la síntaxis

Una D.D.S. utiliza una GLC para especificar la entrada. A cada símbolo de la gramática le asocia un conjunto de atributos y a cada producción un conjunto de reglas semánticas para calcular los valores de los atributos asociados con los símbolos que aparecen en esa producción

Sea n un nodo del A.A.S. que está etiquetado con el símbolo X de la gramática. Se escribe X.a para indicar el valor del atributo a de X en ese nodo (X.a Cuanto vale el atributo del nodo etiquetado con X)

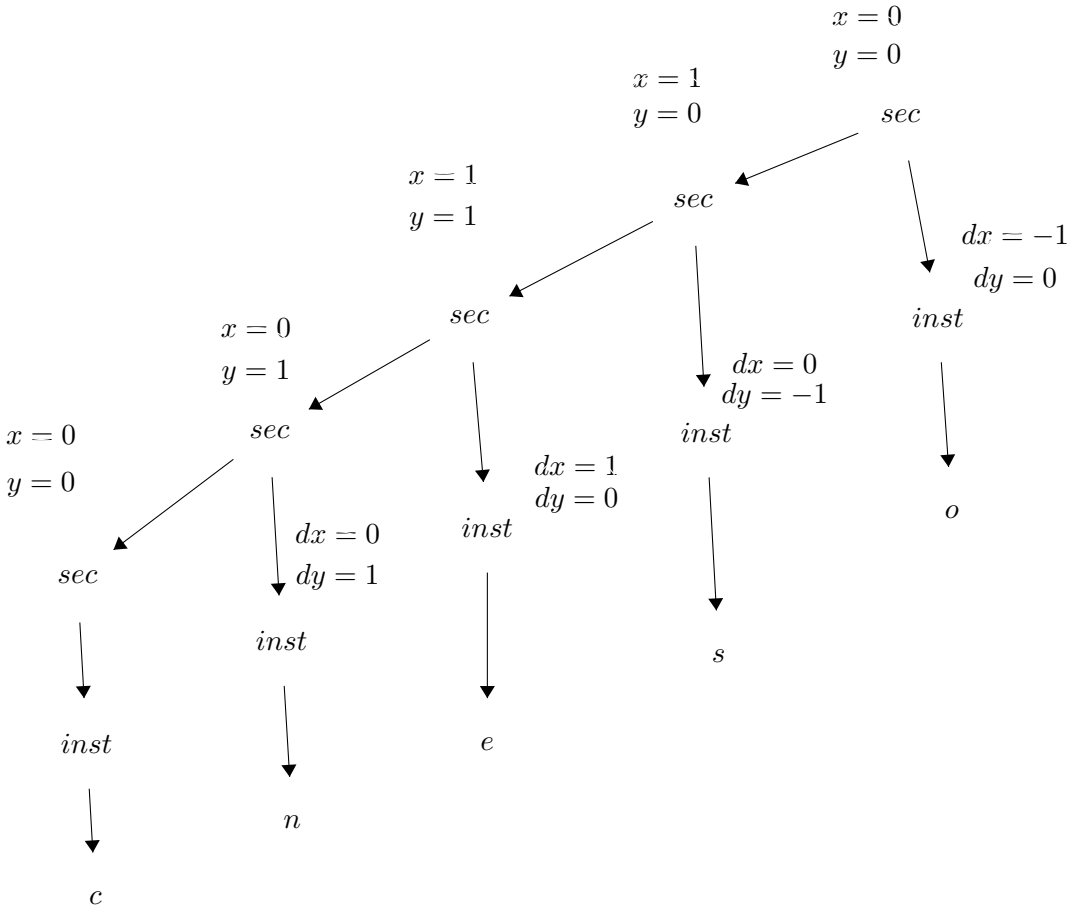
A.A.S. con anotaciones (árbol decorado) Es el A.A.S. que muestra los valores e los atributos en cada nodo.

Atributos sintetizados Aquel cuyo valor en un nodo del A.A.S. se determina a partir de los valores de los atributos de los hijos del nodo.

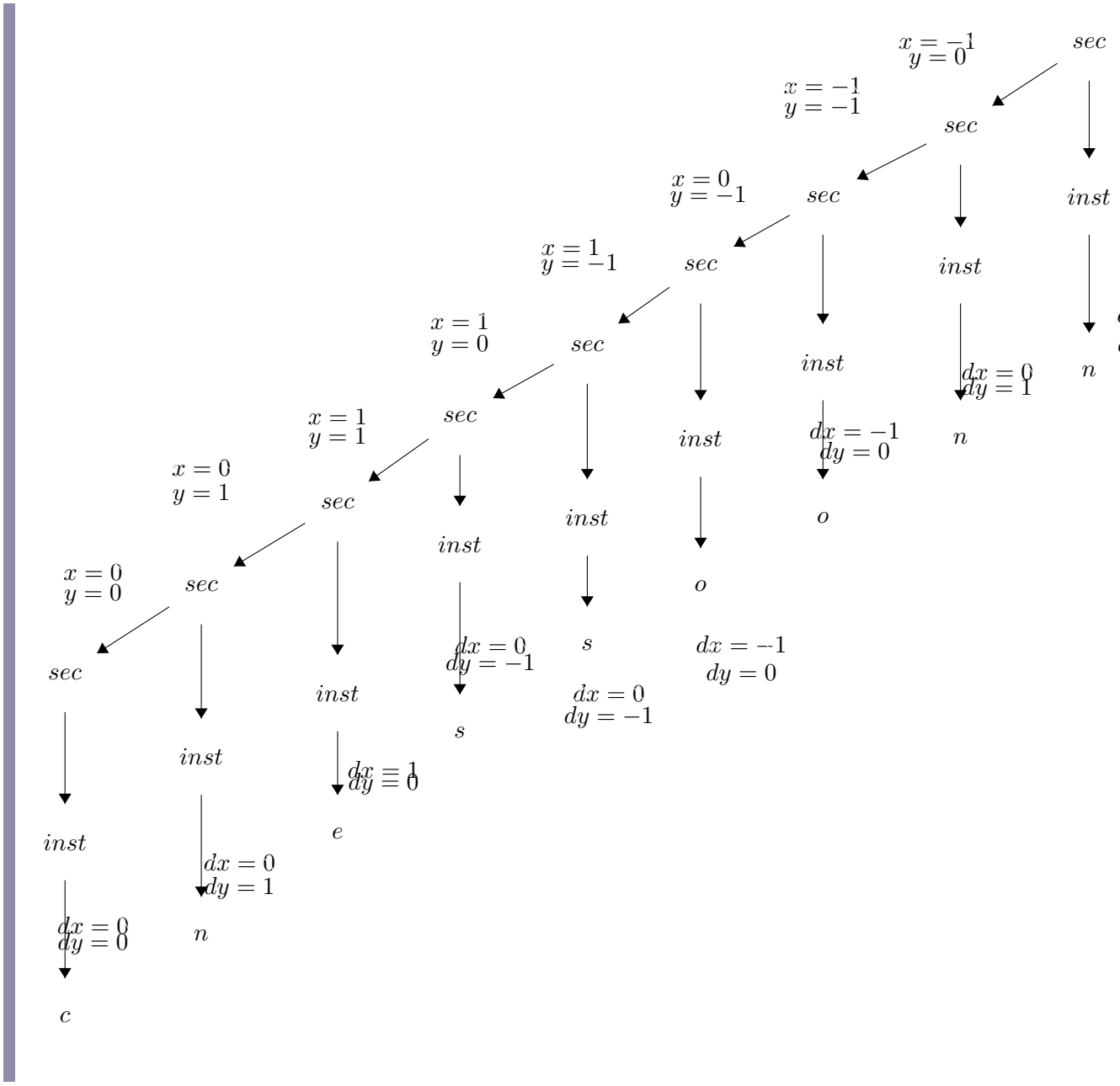
Producción	Regla semántica
exp → exp_1 + ter	exo.t := exp_1.t || ter.t || '+'
exp → exp_1 - ter	exp-t := exp_2.t || ter.t || '-'
exp → ter	exp.t := ter.t
ter → 0	ter := ter:='0'
ter → 1	ter := '1'
...	
ter → 9	ter := '9'

9 - 5 + 2

Producción	Regla semántica
sec → comienza	sec.x := 0 sec.y := 0
sec → sec_1 inst	sec.x := sec_1.x + inst.dx sec.x := sec_1.y + inst.dy
inst → este	inst.dx := 1 inst.dy := 0
inst → norte	inst.dx := 0 inst.dy := 1
inst → oeste	inst.dx := -1 inst.dy := 0
inst → sur	inst.dx := 0 inst.dy := -1



Example 1.7 ()
cnessoonn



A cada símbolo de la gramática se le asigna un conjunto de atributos, incluso el atributo podría ser vacío. Además a cada producción de la gramática se le asigna un conjunto de reglas semánticas, las cuales son utilizadas para calcular los valores de los atributos de los símbolos gramaticales que aparecen en la producción

1.5.3 Recorridos en profundidad

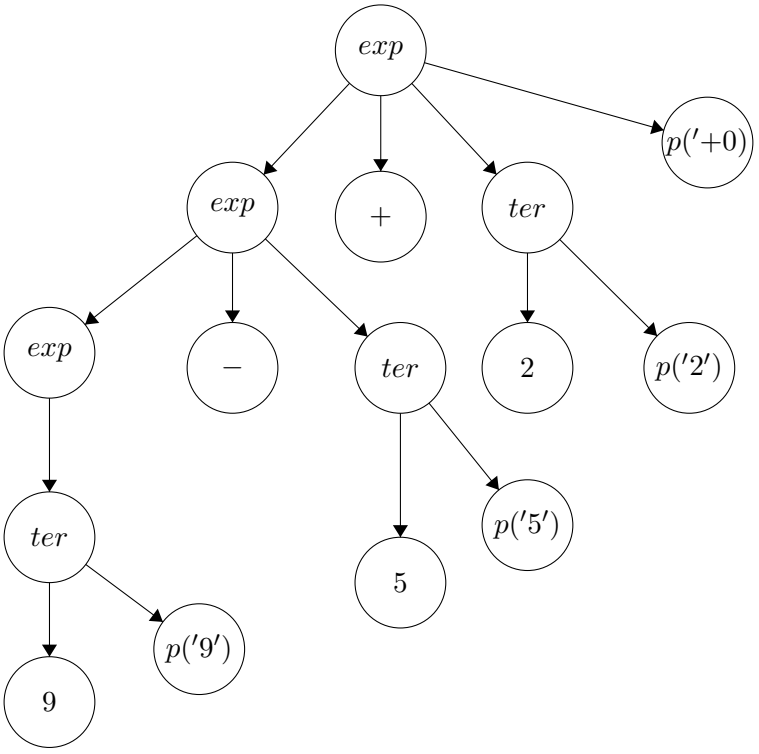
```
1 void visita(Nodo n){
2   //Por cada hijo n de izquierda a derecha de
3   visita(n);
4   //evalua las reglas ásemnticas en el nodo n.
5 }
```

1.5.4 Esquema de traducción

Es una GLC en la que se encuentras intercalados en los lados derechos de las producciones fragmentos de programa llamados acciones semánticas.

1. Primero debemos construir el árbol de análisis sintáctico
2. Pronemos las acciones semánticas donde deban ir en el árbol
3. Recorremos el árbol en profundidad

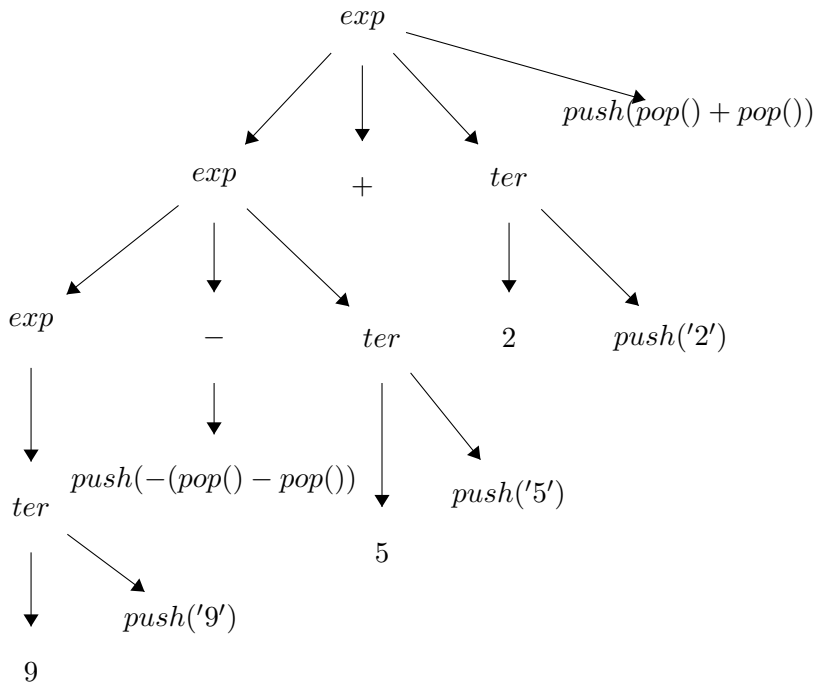
Example 1.8 ()
exp → exp + ter print('+')
exp → exp - ter print('-')
exp → ter
ter → 0 print('0')
...
ter → 9 print('9')



Cuando es dirigida por la sintaxis decorábamos el árbol, la traducción aparecía en la raíz del árbol

El esquema de producción, hace la traducción de infijo a posfijo

Example 1.9 ()
exp → exp . ter + push(pop() + pop())
exp → exp . ter - (push(-(pop() - pop())))
exp → ter
exp → 0 push(p,0)
...
exp → 9 push(p,9)
//Lo que hay entre llaves es llamado acción semántica



El esquema de traducción, sirve para traducir

Semántica: Un fragmento de programa

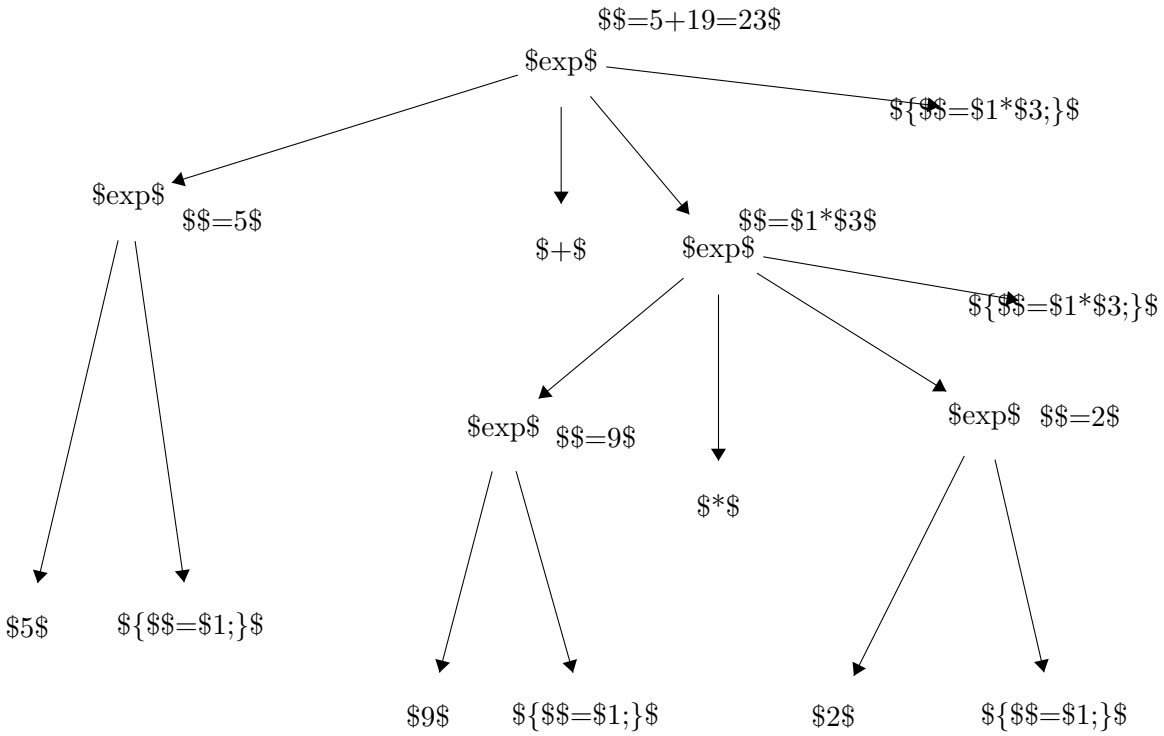
¿Cual es el posfijo de una suma? t significa el posfijo Por tanto podemos decir que el posfijo de una suma es el posfijo del primer operando concatenado con el posfijo del segundo operando concatenado con el operador

Declaraciones
%% sección de reglas gramática %%
codigo de soporte

Especificacion de yacc → *Compilador* → *lenguaje C*

```
/** Archivo para una calculadora hecha en YACC**/

//Lo que hay entre llamaes es la óaccin ásemntica
//Ya que es un fragmento de ócdigo.

//Inicio declaraciones
%{
  //librerias
  #include<stdio.h>
  #define YYSTYPE double
  //S es de stack
%}
%token NUMBER
//íAqu se asigna la precedencia de operadores
%left '+' '-'
%left '*' '/'
//Fin declaraciones

//Inicio óseccin de reglas
//$1 hace referencia al primer ísmbolo gramatical del lado derecho de la
    óproduccin
%%
  list:
  | list '\n'
  | list expr '\n' {printf("\t%.8g\n",$2);}
  ; //Indicamos que se terminaron las producciones de un terminal

  exp: /**los : es nuestra "flecha "**/ NUMBER {$$=$1;}
    |exp'+'exp{$$=$1+$3;}
    |exp'-'exp{$$=$1-$3;}
    |exp'*'exp{$$=$1*$3;}
    |exp'/'exp{$$=$1/$3;}
    |(exp){$$=$2;}
    ;
    //El valor se almacena en pesos, pesos ($$)
    //$$ El valor donde la óproduccin se guarda
    //Se asigna el resultado de la óevaluacin de las expresiones
    //Las expresiones son unas "nerds" requieren ser evaluadas
%%
//Fin declaraciones
void main(){
  yyparse();//parse = explorar
}

int yylex(){
  int c;
  while((c=getchar()) == ' ' || c = '\t');
  if(c == EOF)
    return 0; //No hay áms tokens
  if(c == '.' || isdigit(c)){
    ungetc(c,stdin);
    scanf("%lf", &yylval);
    return NUMBER;
  }
  return c;
}

void yyerror(char *s){
  puts(s);
}
```

Example 1.10 ()
Árbol de análisis sintactico del HOC 1
5 + 9 * 2



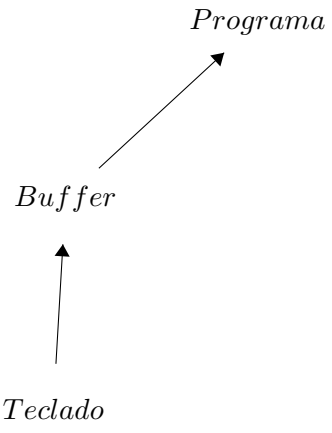
1.6 un poco de YACC

YACC (Yet Another Compile of Compilers) Está compuesto por 3 partes, la primera son delaraciones, la segunda es la sección de reglas, la cual va entre símbolos de porcentaje, aquí se escribe nuestra gramática. La última parte es el código de soporte

```
1  %{
2  #include<stdio.h>
3  //define define una macro = algo que puedo sustituir
4  //Preprocesador sustituye las macros cuando las encuentra
5  #define YYSTYPE double
6  #}
7  %TOKEN number
8  //%left -> tipo de asociatividad
9  %left '+' '-'
10 %left '*' '/'
11
12 %%
13 list: /*Nada*/
14     | list '\n'
15     | list exp '\n' {printf("\ty...8g\n", $2);}
16     ;
17 exp: NUMBER {$$=$1;}
```

```
18      | exp '+' exp {$$=$1+$3;}
19      | exp '-' exp {$$=$1-$3;}
20      | exp '*' exp {$$=$1*$3;}
21      | exp '/' exp {$$=$1/$3;}
22      | '('exp')' {$$ = $2;}
23      ;
24 %%
25 void main(){
26     yyparse();
27 }
28
29 int yylex(){
30     int c;
31     //Este ciclo ignora los espacios en blanco (salta blancos)
32     while((c = getchar()) == '\t' || c=' '); //Enunciado nulo = porque no hace
           nada
33         if(c == EOF)
34             return 0;
35         if(c == '.' || isdigit(c)){
36             mgetc(c,stdin);
37             scanf("%lf", &yylval);
38             return NUMBER; //Retornamos el tipo de token
39         }
40         return c;
41 }
42
43 void yyerror(char *s){
44     puts(s);
45 }
```

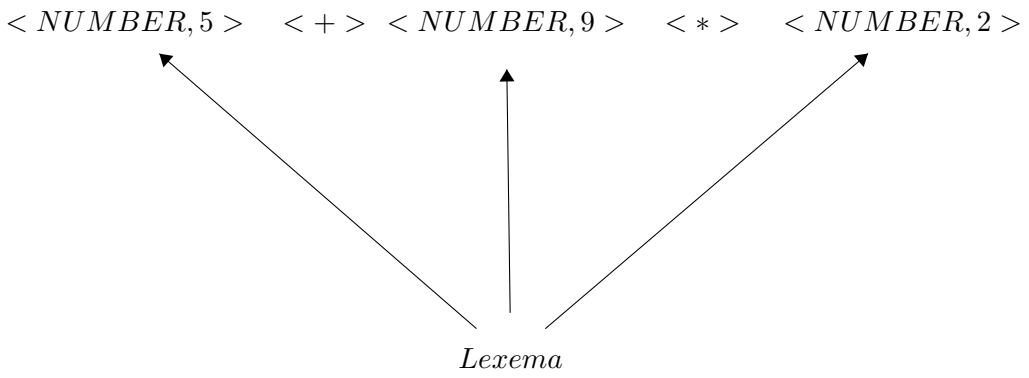
Recordemos que la entrada de datos se realiza de la siguiente manera



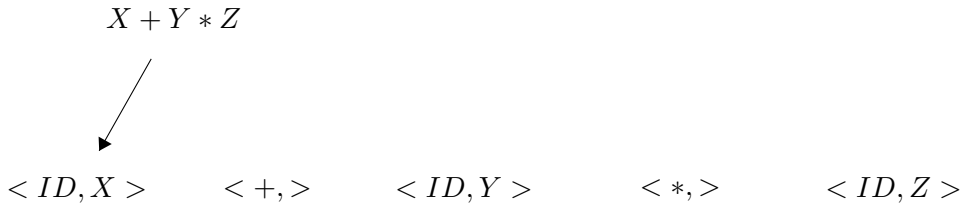
yylex() Divide en tokens una cadena

Example 1.11 ()
Supongamos que tenemos la siguiente expresión
5 + 9 * 2
yylex entonces realiza
<NUMBER><+><NUMBER><*><NUMBER>
Una ves que se realiza esto hay que saber que cosa es NUMBER

<NUMBER,5><+><NUMBER,9><*><NUMBER,2>



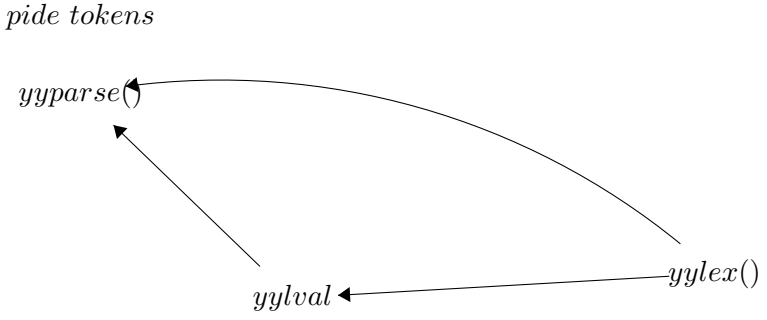
Por tanto yylex lo que haría es



ungetc(c, stdin) nos sirve para leer el número completo que se ha ingresado

yylex() se comunica con yyparse() retornando un valor de tipo entero o utilizando una variable global

yylval es del mismo tipo de los elementos en la pila de Yack




```
1 %{
2     double memm[26]; //Tabla de signos patito
3 %}
4 %union{ // En lugar de #define YYSTYPE
5     double val;
6     int index;
7 }
8 %token<val> NUMBER //TOken
```

```
%token <index> VAR //Token
%type<val> exp //ísmbolo gramatical no terminal
%right '='//Asociatividad por la derecha
...
%left UNARYMINUS
%%
...
exp: ...
    | VAR '=' exp{$$=[1]=$3;} //Una expresion puede ser una variable
    | VAR {$$ = mem[1];}
...
    | exp '/' exp{
        if($3 == 0.0)
            puts("division por cero");
        $$ = $1 / $3;
    }
...
    | '-' exp UNARYMINUS{$$ = -$2;}
    ;
%%
...
int yylex(){
    int c;
    ...
    //Identifica los numeros
    if(c == '.' || isdigit(c)){
        ...
        scanf("%lf", &yylval.val);
        ...
    }
    //islower es una macro con parametro
    //Identifica las variables (letra úminscula)
    if(islower(c)){
        //c es un lexema
        //yyval = UNION
        yylval.index = c-'a';
        return VAR; //Tipo de token
    }
    ...
}
```

1.7 Análisis sintático descendente

Las construcción descendente del A.A.S se hace empezando por la raiz y realizando de forma repetida los 2 pasos siguientes.

1. En el nodo n etiquetado con el no terminal A, seleccionese una de las producciones para A y construyanse los hijos de n para los símbolos del lado derecho de la producción

2. Encuentrese el siguiente nodo en el que ha de construirse un subárbol.

Example 1.12 ()
tipo → id
| ↑ id
| array[simple] of tipo

simple → integer
| char
| num puntopunto num

```
1  /**Verifica que el token actual sea igual al que sigue**/
2  void parea(Token t){
3      //preana es una variable global
4      //preana guarda el token actual
5    if(preana == t)
6        //Se obtiene el siguoente componente lexico
7      preana = sigcomplex(); //en yacc sigcomplex = yylex
8    else error();
9  }
10
11 /**Para escribir el ócdigo nos basamos en lo
12 que hay en el lado derecho de la óproduccin tipo**/
13 void tipo(){
14     if(preana == INTEGER ||
15      preana == CHAR ||
16      preana == NUM)
17      simple();
18     if(preana == '$\uparrow$'){
19        //El token actual es una flecha
20        //Comparamos el token actual
21      parea('$\uparrow$');
22      //Despues de una flecha hay un ID
23      parea(ID);
24     } else if(preana == ARRAY){
25      parea(ARRAY);
26      pareana ('[');
27      simple();//Es un no terminal
28      parea(']');
29      parea(of);
30      tipo();
31     } else error();
32
33 }
34
35
36 /**Para escribir el ócdigo nos basamos en lo
37 que hay en el lado derecho de la óproduccin simple**/
38 void simple(){
39    if(preana == INTEGER)
40      parea(INTEGER);
41    else if(preana == CHAR)
42      parea(CHAR);
43    else if(preana == NUM){
44      parea(NUM);
45      parea(PUNTOPUNTO);
46      parea(NUM);
47    } else error();
48 }
```

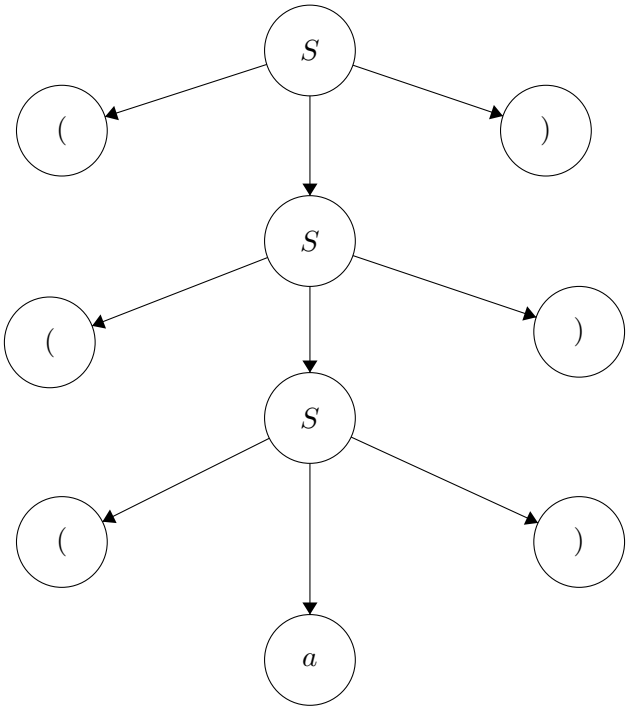
HOC 3 Del programa HOC 3 tenemos lo siguiente

$T = \{integer, char, \uparrow, id, [,], of, array, num, puntopunto\}$
$N = \{simple, tipo\}$
$S = \{tipo\}$

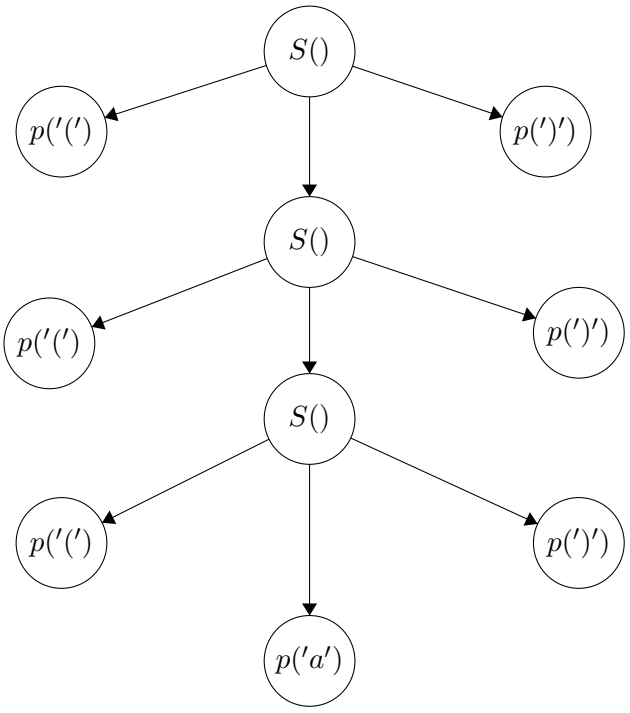
Example 1.13 ()
S $\rightarrow$ | a
Código

```
1    void S(){
2      if(preana == 'a')
3        parea('a');
4      else if(preana == '('){
5        parea('(');
6        s();
7        parea(')');
8      } else error();
9    }
```

probamos el código con la cadena (((a))) con lo que se obtiene el siguiente árbol sintáctico



El árbol de llamadas es el siguiente



El análisis lo estamos haciendo de forma implícita.
En YACC la sección de reglas es la siguiente

```
%%
S:    'a'
    | '(' s ')'
    ;
%%
```

1. Expresar el problema de forma recursiva
2. Parar recursividad (caso base)
3. llamada recursiva
4. Probar la recursividad

1.8 Análisis sintactico predictivo recursivo

Análisis sintáctico predictivo Se basa en el token actual. Seleccionar que producción utilizaremos en base al token actual.

Es una análisis sintáctico donde la cadena de entrada se analiza con un conjunto de procedimientos recursivos y para cada no terminal de la gramática se escribe un procedimiento

Es un método descendente en el que se ejecuta un conjunto de procedimientos recursivos para procesar la entrada. A cada no terminal de una gramática se le asocia un procedimiento.

Esto dependerá de que la gramática sea recursiva

1.9 PRIM(α)

Es el conjunto de tokens que opere como primeros símbolos de una o mas cadenas generadas a partir de α

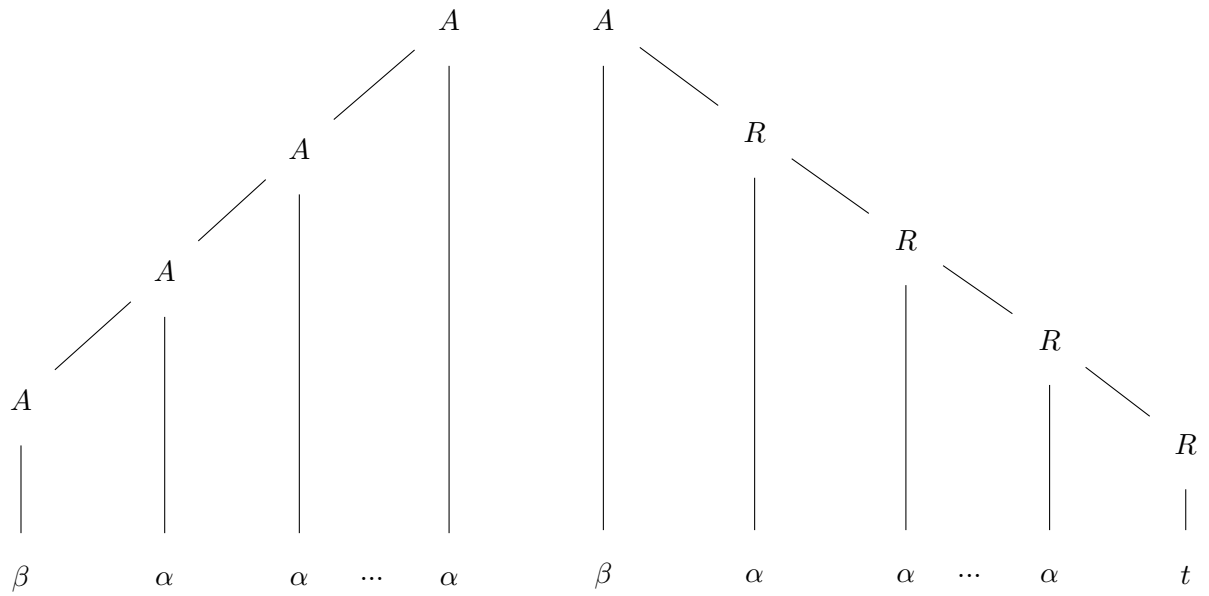
PRIM(simple)={integer, char, num}
PRIM(↑id) = {↑}
PRIM(array[simple] of tipo) = {array}
Si hay 2 producciones. A $\rightarrow \alpha$ y A $\rightarrow \beta$
PRIM(α) $\cup$ PRIM(β) = $\varnothing$

Example 1.14 ()
SI A $\rightarrow$ a y A $\rightarrow$ b
PRIM(a) $\cap$ PRIM(b) = {a} $\cap$ {b} = $\varnothing$

1.10 Recursividad por la izquierda

expo $\rightarrow$ exp + ter
A $\rightarrow$ Aα | β
A $\rightarrow \beta$R
R $\rightarrow \alpha$R | ϵ

Cuando el no terminal de la producción aparece del lado derecho de la producción como el primer símbolo gramatical podemos decir que hay recursividad por la izquierda



exp→exp + ter

¿Qué es alpha y beta? Una cadena de símbolos gramaticales = a un terminal o un no terminal
Que condición deben cumplir PRIM(α) y PRIM(β)? Que la intersección nos dé conjunto vacío

Esquema de traducción: Convierte de infijo a posfijo exp → exp + ter{print('*')}
exp → exp - ter{print('-')}
exp → ter
ter → 0 {print('0')}
ter → 1 {print('1')}
.
.
.
ter → 9 {print('9')}
A → Aα | Aβ | γ
A → γR
R → αR | βR | ϵ
exp → exp + ter{print('+')}
exp → exp - ter{print('-')}
exp → ter
ter → 0{print('0')}
ter → 1{print('1')}
.
.
.
ter → 9{print('9')}
Su equivalencia al sin recursividad es
exp → terR
R → +term{print('+')}R
R → -term{print('-')}R
R → ϵ
ter → 0{print('0')} ter → 1{print('1')} .
.
.
ter → 9{print('9')}

Example 1.15 ()
S → aABe
A → b
B → d

Elementos
T = {a,b,d,e}
N = {S,A,B}
S={S}

```
  void S(){
  if(preana == 'a'){
    parea('a');
    A();
    B();
    parea('e');
  } else  {
    error();
  }
}

void A(){
  if(preana == 'b'){
    parea('b');
  } else {
    error();
  }
}

void B(){
  if(preana == 'd'){
    parea('d');
  } else {
    error();
  }
}
```

Que decimos de dos gramáticas que generan el mismo lenguaje? Son gramáticas equivalentes.

Laboratorio Martes, laboratorio de programación 2

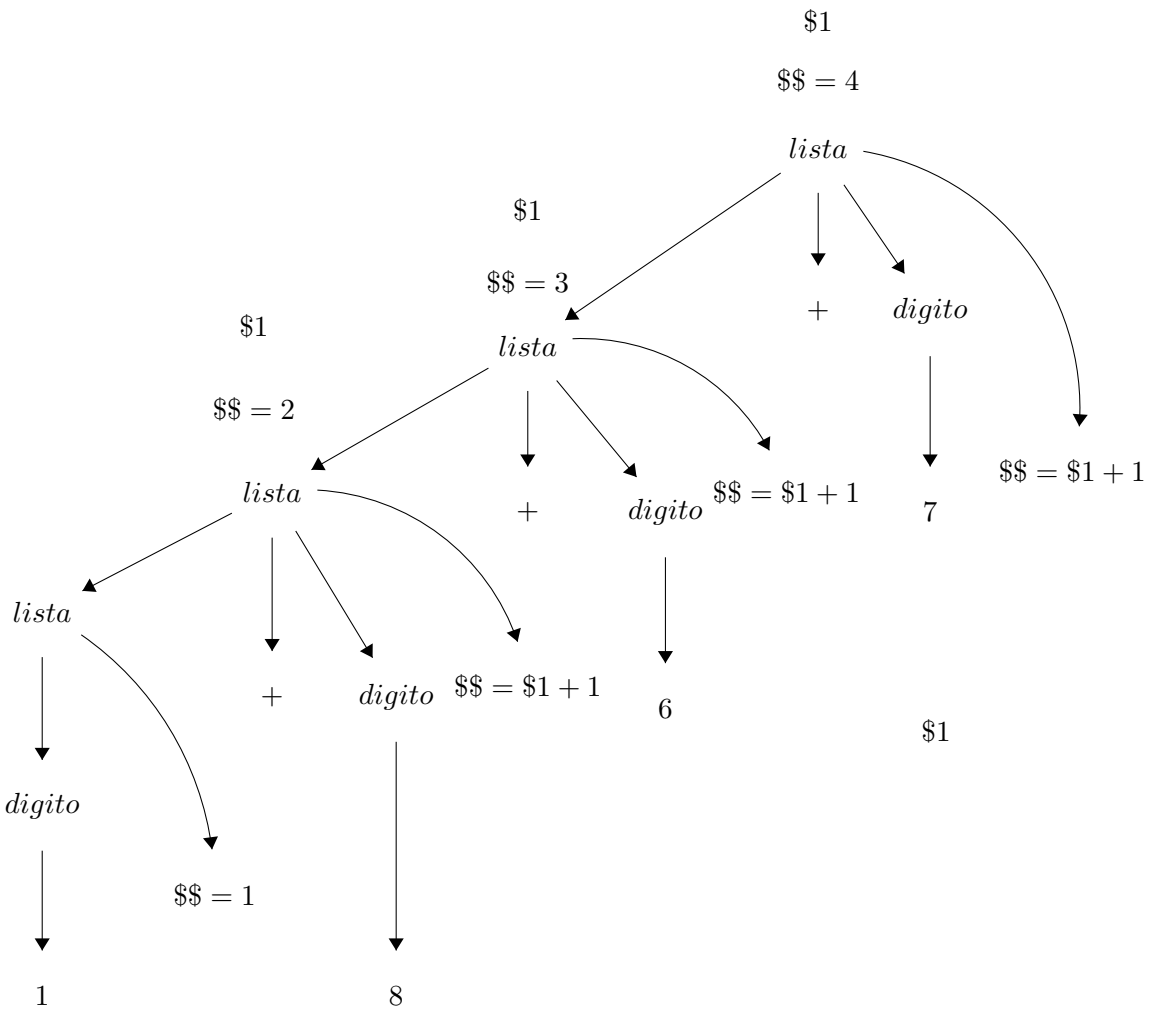
```
void s(){
  if(preana == '('){
    parea('(');
    S();
    parea(')');
  } else ; //enunciado nulo
}
```

Example 1.16 ()
Traducir la gramática a YACC
lista → digito
lista → lista + digito
digito → 0 | 1 | 2 | ... | 8 | 9

```
  %%
  lista: digito {$$ = 1}
  | lista '+' digito {$$ + $$ + 1}
  ;
  digito: '0'
  |   '1' {}
  |   '2' {}
  .
```

```
9     .
10    .
11    |     '9' {}
12    ;
13  %%
```

Diagrama

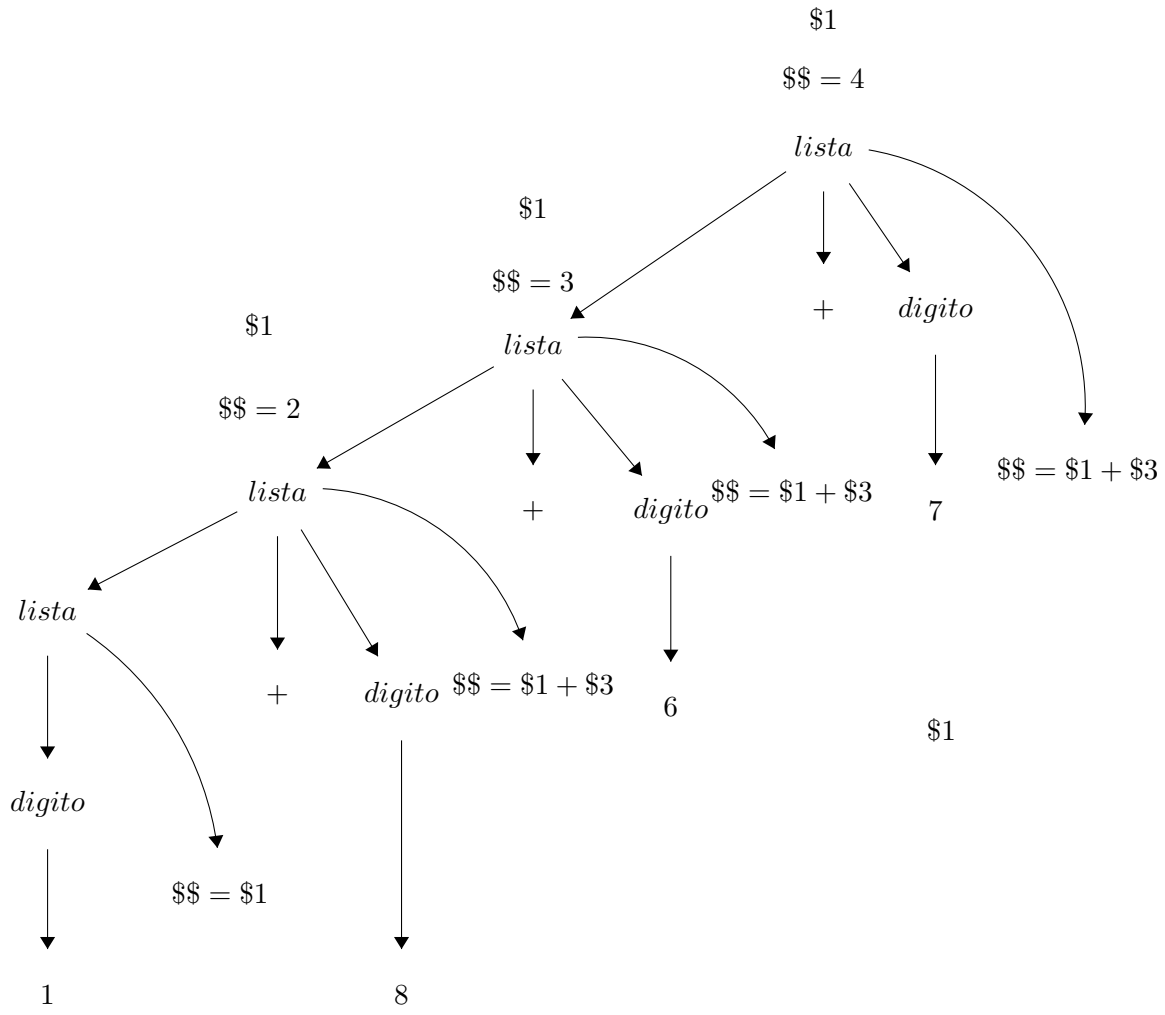


Example 1.17 ()
Que pasa si del ejemplo anterior deseamos sacar la suma

```
1   %%
2   lista: digito {$$ = $1;}
3   | lista '+' digito {$$ = $1 + $3;}
4   ;
5   digito: '0'
6   |     '1' {$$ = 0;}
7   |     '2' {$$ = 1;}
8   .
```

```
9     .
10    .
11    |    '9' {$$ = 9;}
12    ;
13 %%
```

El diagrama resultante es:



Si hacemos el analizador sintáctico y léxico a mano obtenemos

```
1
2  int preana;
3
4  void S(){
5
6    if(preana == '('){
7      parea('(');
8      S();
9      parea(')');
10   } else if(preana == 'a'){
11     parea('a');
12   } else error();
```

```
13 }
14
15 int sigcomplex(){
16   return getchar();
17 }
18
19 void error(char *s){
20   puts("sintax error");
21 }
22
23 void main(){
24   preana = sigcomplex();
25   S();
26 }
```

1.11 Un par de programas

1.12 Notacion

1.12.1 Terminales

- Las primeras letras minus del alfabeto como a, b, c
- Los símbolos de operador como +, -, *, etc-
- Los símbolos de puntuación, como paréntesis, coma, etc.
- Los dígitos 0, 1, ..., 9
- Cadenas en negritas como id o if

1.12.2 No terminales

- Las primeras letras mayúsculas del alfabeto como A, B, C
- La letra S (símbolo inicial)
- Los nombres en cursivas minúsculas

1.12.3 Símbolos gramaticales

Últimas letras mayúsculas del alfabeto como X, Y, Z

1.12.4 Cadenas de símbolos gramaticales

Letras griegas minúsculas α, β, γ

1.12.5 Si A $\rightarrow \alpha_1$, A $\rightarrow \alpha_2$, ..., A $\rightarrow \alpha_n$

Son todas las producciones con A a la izquierda se puede escribir

A $\rightarrow \alpha_1 \mid \alpha_2 \mid ... \mid \alpha_n$

α_1, α_2, ..., α_n se denominan alternativas de A

1.12.6 Símbolo inicial

A menos que se diga otra cosa el lado izquierdo de la primer producción es el símbolo inicial

1.13 Derivación

Se dice que $\alpha A \beta \Rightarrow \alpha\gamma\beta$. Si A $\rightarrow \gamma$ es una producción

Si $\alpha_1 \Rightarrow \alpha_2 \Rightarrow ... \Rightarrow \alpha_n$ se dice que α_1 deriva α_n

EL símbolo $\Rightarrow$ significa "deriva de un paso"

$* \Rightarrow$ deriva en cero o más pasos

Asi

1. $\alpha * \Rightarrow \alpha$
2. Si $\alpha * \Rightarrow \beta$ y $\beta \Rightarrow \gamma$ entonces $\alpha * \Rightarrow \gamma$

$+ \Rightarrow$ deriva en uno o más pasos

L(G) es el lenguaje generado por la gramática G

Las cadenas de L(G) pueden tener solo símbolos terminales de G

Una cadena w está en L(G) si y solo si S $+ \Rightarrow$ w. A w se le llama frase de G

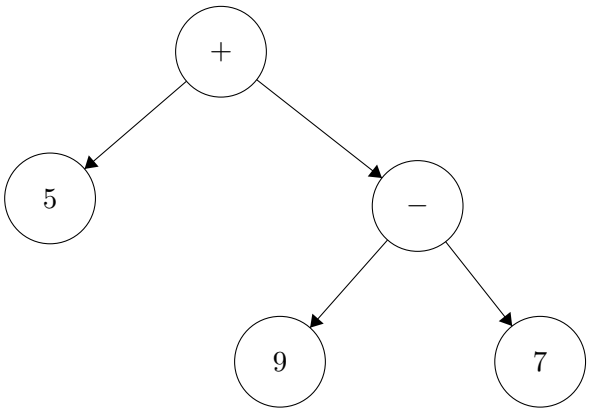
Si S $* \Rightarrow \alpha$ dónde α puede contener no terminales entonces se dice que α es una forma de frase de G

$\alpha \Rightarrow \beta$ deriva por la izquierda

$\alpha \Rightarrow \beta$ deriva por la derecha

Si S $\Rightarrow \alpha$ entonces se dice que α es una forma de frase izquierda

Example 1.18 (R)
etomemos el ejercicio anterior
5 + 9 - 7 pasa a ser 5 9 + 7 -



Árbol sintáctico ≠ Árbol de análsis sintáctico

```
%%
  exp: exp '+' ter{$$ = creaNodo("+", $1, $3);}
  |  exp '-' ter{$$ = creaNodo("-", $1, $3);}
  |  ter {$$ = $1;}
  ;
  ter: '0'  {$$ = creaNodo("0", NULL, NULL);} //{$$ = $1;}
  |  '1'  {$$ = creaNodo("1", NULL, NULL);} //{$$ = $1;}
  |  '2'  {$$ = creaNodo("2", NULL, NULL);} //{$$ = $1;}
  |  '3'  {$$ = creaNodo("3", NULL, NULL);} //{$$ = $1;}
  .
  .
  .
  |  '9'  {$$ = creaNo1do("9", NULL, NULL);} //{$$ = $1;}
  ;
%%
```

En C obtenemos lo siguiente

```
int yylex(){

  char cad[2];

  if(isdigit(s)){
    cad[0] = c;
    cad[1] = 0;
  }
}

yyval = creaNodo( strdump(cad), NULL, NULL)
```

Para el de vectores

Example 1.19 ()
Utilizando [5 6 8]
[7622] + [3516555]
<[><NUMBER,5><NUMBER,6><NUMBER,8><]>
<VECTOR, [5 6 8]>
<VECTOR, [7, 6, 22]><+,><VECTOR, [35, 16, 555]>

1.14 NOTACIÓN

1. Terminales
2. No terminales
3. Símbolos gramaticales
4. Cadenas de terminales las útlimas letras minpusculas del alfabeto principalmente u, v, ..., z
5. Cadenas de símbolos gramaticales
6. A $\alpha_1, ..., A \to A_n$ se abrevia $A \to \alpha_1|\alpha_2|...|\alpha_n$
7. Símbolo inicial

Se dice que $\alpha A\beta \Rightarrow \alpha\gamma\beta$ Si $A \Rightarrow \gamma$ es una producción si $\alpha_1 \Rightarrow \alpha_2 \Rightarrow ... \Rightarrow \alpha_n$ se dice que α_1 deriva α_n

Se símbolo $\Rightarrow$ significa "deriva en un paso"

$* \Rightarrow$ deriva en cero o más pasos. Así

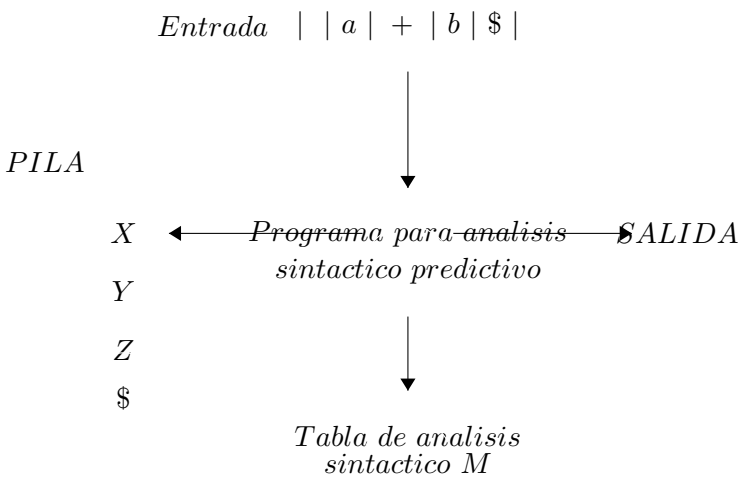
1. $\alpha* \Rightarrow \alpha$
2. Si $\alpha* \Rightarrow \beta$ y $\beta* \Rightarrow \gamma$

$+ \Rightarrow derivaenunoomáspasos$

L(G) es el lenguaje generado por la grmática G
Las cadenas de L(G) pueden contener solo símbolos terminales de G
Una cadena W esta en L(G) si y solo si S+ $\Rightarrow$W. A W se le llama frase de G si S $* \Rightarrow \alpha$ dónde α puede contener no terminales entonces se dice que α es una forma de frase de G
$\alpha_{mi} \Rightarrow \beta$ deriva por la izquierda $\alpha_{md} \Rightarrow \beta$ Deriva por la derecha
Si S $* \Rightarrow \alpha$ entonces se dice que α es una forma de frase izquieda
E $\to$ TE'
E' $\to$ +TE' | ϵ
T $\to$ FT'

T' → *FT' | ϵ
F → (E) | id

	id	+	*	(	)	$
E	E→TE'			E →TE'		
E'		E'→+RE'			E'→ ϵ	E'→ ϵ
T	T→FT'			T'→*FT'		
T'		T'→ ϵ	T'→*FT'		T'→ ϵ	T'→ ϵ
F	F→id			F→(R)		



1.15 Análisis sintáctico predictivo no recursivo

Es una análisis sintáctico que utiliza una pila para realizar análisis sintáctico. En la pila se guardan símbolos gramaticales.

Entrada Una cadena w y una tabla de análisis sintáctico M para la gramática G

1.16 Salida

Si w esta en L(G) una derivación por la izquierda de w si no error

1.17 Método

Al principio el análisis sintáctico tiene $S en la pila con S en la cima y W$ en la entrada. Apuntar ae al primer símbolo de w$

```
1 do{
2   //sea X el ísmbolo de la cima de la pila
3   // y a el ísmbolo apuntando por ae
4   if(X es un token o $){
5     if(X == a){
6       extraer X de la pila y analizar ae;
7     } else error();
8   } else if(M[X, a] == X -> Y1 Y2 ... Yk){
```

```
9       extraer X de la pila
10      meter  Yk, Yk-1, ..., Y1 con Y1 en la cima
11      emitirla produce X -> Y1 Y2...YK
12    } else error()
13  } while(X != $);
```

$E	id$	E→TE’
$E’T	id$	T→FT’
$E’T’F	id$	F→id
$E’T’ id	id$	
$E’T’	$	T’ → ϵ
$E’	$	E’→ ϵ
$	$	

X = E, a = id
X = T, a = id
X = F, a = id
X = id, a = id
X = T’, a = $
X = E’, a = $
X = $, a = $
Resultado E→TE’→FT’E’→idT’E’→idϵE’→id$\epsilon\epsilon$

1.18 Primero α

Es el conjunto de tokens que inician cadenenas derivadas de α si $\alpha* \Rightarrow \epsilon$, entonces ϵ también está en PRIMERO(α).

1.19 SIG(A)

Es el conjunto de tokens "a" que pueden aparecer inmediatamente a la derecha de A en alguna forma de frase i.e., el conjunto de tokens a tal que haya una derivación de la forma $S * \Rightarrow \alpha A a \beta$ (Forma de frase) para algún α y β Si A puede ser el símbolo situado más a la derecha en una forma de frase entonces $ está en SIG(A).

1.20 Cálculo de PRIMERO(α)

Para todos los ímbolos gramaticales X, aplíquense las reglas siguientes hasta que no se puedan añadir más tokens o ϵ a ningún conjunto PRIM

Forma de frase Es una cadena de símbolos gramaticales, que se derivó a partir del símbolo inicial.

- Debe contener mínimo un terminal
- Debe derivar del símbolo inicial

1. Si X es token entonces PRIM(X) = {x}
2. Si $X \rightarrow \epsilon$ es una producción entonces añádase ϵ a PRIM(X)

3. Si X es no terminal $X \rightarrow Y_1 Y_2 ... Y_k$ es una producción, entónces póngase "a" en PRIM(Y_i),..., PRIM(Y_{i-1}) es decir $Y_1 ... Y_{i-1} * \Rightarrow \epsilon$ Si ϵ está en PRIM(Y_j). $\forall_j = 1, 2, ..., k$ entonces añádase ϵ a PRIM(X). Si Y_1 no deriva ϵ, entonces no se añade más a PRIM(X), pero si $Y_1 * \Rightarrow \epsilon$ entonces se le añade PRIM(Y_2) y así sucesivamente

Se puede calcular PRIM para cualquier cadena $X_1, X_2, ..., X_n$ de la siguiente forma: añádase a PRIM($X_1, ..., X_n$) $\forall$ los símbolos distintos de ϵ PRIM(X_1)

Si ϵ está en PRIM(X_1), añádanse también los símbolos distintos de ϵ de PRIM(X_2); y así sucesivamente. Por último añádase ϵ a PRIM($X_1, ..., X_n$) si $\forall_i$ PRIM(X_i) contiene ϵ

1.21 Cálculo de SIG(A)

Para todos los no terminales A. Aplíquense las reglas siguientes hasta que no se puedan añadir nada más a ningún conjunto SIG

1. Póngase $ en SIG(S).
2. Si hay una producción $A \rightarrow \alpha B \beta$ entonces todo lo que este en PRIM(β) excepto ϵ se pone en SIG(B)
3. Si hay una producción $A \rightarrow \alpha B$ o una producción $A \rightarrow \alpha B \beta$, donde PRIM($\beta$) obtenga ϵ (i.e. $\beta * \Rightarrow \epsilon$) entonces todo lo que esté en SIG(A) se pone en SIG(B)

1.22 Construcción de tablas de A.S.

Entrada Una gramática G

Salida La tabla de A.S. M

1.22.1 Método

1. Para cada producción $A \rightarrow \alpha$ de la gramática, dense los pasos 2 y 3
2. Para cada token "a" de PRIM(α) añádase $A \rightarrow \alpha$ a M

$$A, a$$

3. Si ϵ esta en PRIM(α) añádase $A \rightarrow \alpha$ a M

$$a, B$$

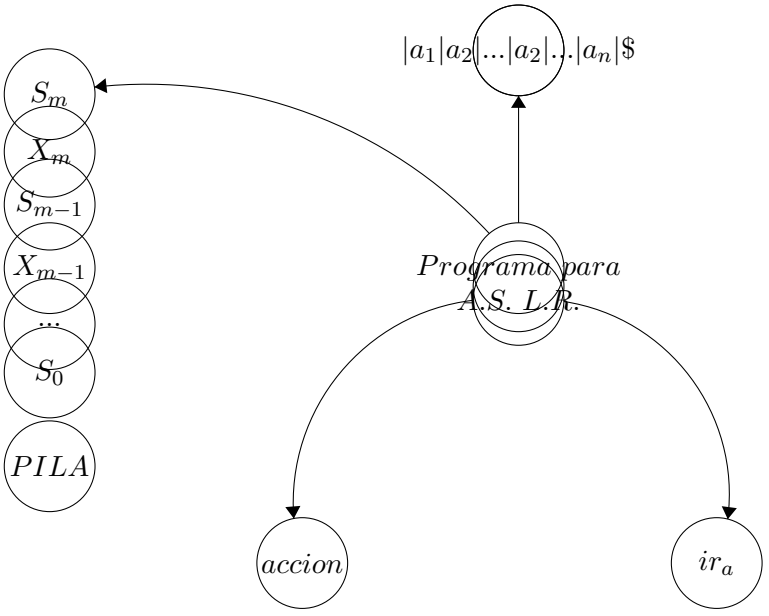
para cada token b de de SIG(A). Si ϵ esta en PRIM(α) y $ esta en SIG(A) añádase $A \rightarrow \alpha$ a M[A,$]

4. Hágase que cada cadena entrada no definida de M sea error

Example 1.20 ()
$E \to TE'$
$E' \to +TE'|\epsilon$
$T \to FT'$
$T' \to *FT'|\epsilon$
$F \to (E)|id$

Example 1.21 ()
PRIM(F) =
F $\to E$
$X \to Y_1, Y_2, Y_3$
$PRIM((E)) = PRIM(($
$F \to id$
$PRIM(id) = id$

1.23 Análisis sintáctico LR



A.S. LR

Entrada Una cadena w y una tabla de A.S. LR con las funciones acción e ir_a para la gramática G

Salida Si w esta en L(G), un anális sintáctico ascendente de w si no error

Método Inicio S_0 está en la pila del análisis sintáctico y W$ esta en el buffer de entrada

1.24 Pasos

1. di significa desplazar y meter en la pila el estado i
2. r, significa reducir por la producción número J
3. acep significa aceptar
4. Espacio en blanco significa error

Apuntar ae al primer símbolo de w$ repetir.

Sea S el estado en la cima d ela pila y a en el símbolo por ae
SI acción

$$S, a$$

= desplazar S' entonces
-meter a y después S' en la cima
-avanzar ae al siguiente símbolo de entrada SI_NO SI acción

$$S, a$$

= reducir A $\rightarrow \beta$ entonces
-sacar 2*$|\beta|$ símbolos de la pila
-Sea S' el estado en la cima de la pila A y después ir_a

$$, A$$

-Emitir la producción A$\rightarrow \beta$
SI_NO
SI accion

$$S, a$$

= aceptar entrada
fin con exito SI_N0
error
FIN_REPETIR

1. E $\rightarrow$ E + T
2. E $\rightarrow$ T
3. T $\rightarrow$ T * F
4. T $\rightarrow$ F F
5. F $\rightarrow$ (E)
6. F $\rightarrow$ id

	id	+	*	(	)	$	E	T	F
0	d_5			d_4			1	2	3
1		d_6				accep			
2		r_2	d_7		r_2	r_2			
3		r_4	r_4		r_4	r_4			
4	d_5			d_4			8	2	1
5		r_6	r_6		r_6	r_6			
6	d_5			d_4				9	3
7	d_5			d_4				9	3
8		d_6			d_{11}				
9	r_1	d_7		r_1	r_1				
10		r_3	r_3		r_1	r_1			
11		r_5	r_5		r_5	r_5			

Example 1.22 ()
id$ Simulando la gramática escrita anteriormente

1. a = id, S = 0
2. a = $, S = 5
3. S' = 0 A = F
4. a = $, S = 3
5. S' = 0, A = T
6. a = $, S = 2
7. S' = 0, A = E
8. a = $, S = 1

PILA	ENTRADA	SALIDA
0	id$	
0id5	$	F→id
0F3	$	T→F
0T2	$	E → T
0E1	$	acep

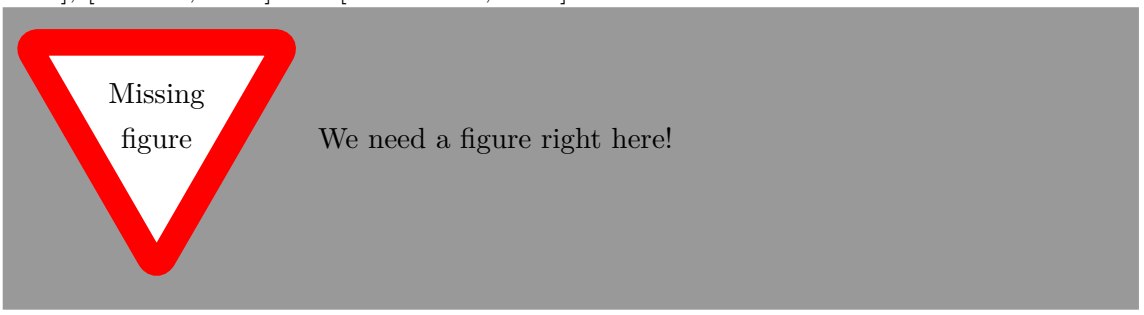
Example 1.23 ()
Probemos ahora con id + id$

Chapter 2

Segundo parcial

Here is chapter 2. If you want to leearn more about LaTeX 2_ε, have a look at [Madsen, 2010], [Oetiker, 2010] and [Mittelbach, 2005].

I think this wor
pelled



Chapter 3

Conclusion

In case you have questions, comments, suggestions or have found a bug, please do not hesitate to contact me. You can find my contact details below.

Jesper Kjær Nielsen
jkn@es.aau.dk
http://kom.aau.dk/~jkn
Niels Jernes Vej 12, A6-309
9220 Aalborg Ø

Bibliography

Madsen, L. (2010). Introduktion til LaTeX. http://www.imf.au.dk/system/latex/bog/.

Mittelbach, F. (2005). *The LATEX companion*. Addison-Wesley, 2. ed. edition.

Oetiker, T. (2010). The not so short a introduction to LaTeX2e. http://tobi.oetiker.ch/lshort/lshort.pdf.

Appendix A

Appendix A name

Here is the first appendix