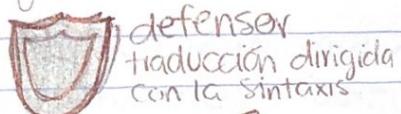


# Compiladores



## Background

expresiones regulares  
lenguajes regulares  
automatas finitos  
tabla hash  
análisis sintáctico  
gramáticas libres de contexto  
automata de pila  
forma norma G.  
quitar E de la gramática.



espada  
LALR

Varios ejemplos

Anexos  
Java → C

Un poco de historia  
el primer compilador de C++ hacia el siguiente proceso

C++ → compilador → C → compilador → binario

JS → compilador avanzado → HTML ← también le llaman traspilación alto nivel a alto nivel

Si en mayo esta terminado el proyecto y se presenta en expo escam - tienes 10



## Compiladores

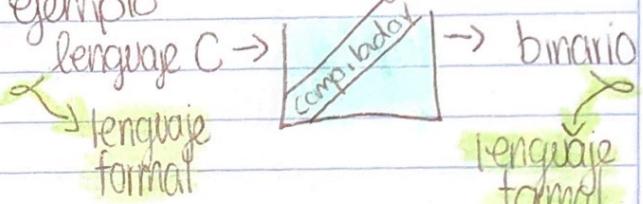
(principios técnicas y herramientas)

Alfred V. Mónica S.  
Ravi Sethi

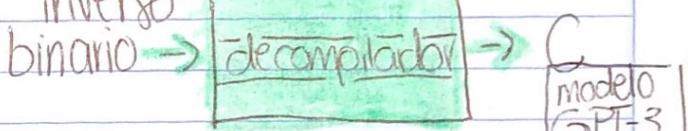
¿Qué es un compilador?

traductor de lenguajes formales

ejemplo



Existen también los decopiladores que hacen el proceso inverso



algo similar

lenguaje natural → intérprete  
(versión formal)

ej. suma dos y tres R=cinco

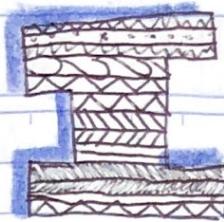
¿Se puede el proceso inverso?

lenguaje formal  
lenguaje C → compilador → l.



(lenguaje natural debe ser en versión formal)

natura  
versión foma  
JEAN BOOK

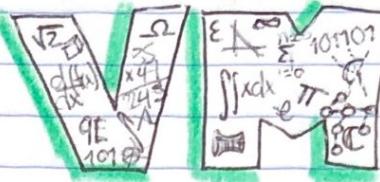


# Interprete

## Compilador

según libro "programa que puede leer un programa en un lenguaje (fuente) y traducirlo en uno equivalente en otro lenguaje".  
 $f(x)$ : reportar errores en el código fuente que detecte en el proceso de traducción

VS



Libro AHO

ejecuta lo que te diga sin traducción

## genera un programa ejecutable

.C → ← → .OUT - Linux  
 .exe - Windows

Sync hay lenguajes compilados ↔ interpretados ↗

esto es que en alguna parte del proceso se compila y en otra se interpreta → ej. PHP  
 en el pasado

hola.php ---> hola.html  
 servidor  
 sistema.php

Siguiendo el principio de la definición de forma literal podríamos decir que está compilando .php a .html por otra parte en el lado del servidor no se queda un archivo  
 ∴ podríamos decir que está interpretado.

## SOLUCIÓN

"almacenar 1 versión pre-compilada"

hola.php --- sistema PHP ---> hola.html

guardo hola.phpc  
 JEAN BOOK  
 compilo 1 vez

Interpreto varias veces

## Interprete

procesador de lenguaje que no genera un programa de traducción, este da la capacidad de ejecutar directamente las operaciones especificadas en el programa de origen con las entradas fixas: ofrece mejores diagnósticos de los resultados es una acción.

</> no hay lenguaje interpretado o compilado

ej. python usa un intérprete para esto, más se le pudo ocurrir hacer un compilador de python

\* si ejecuta la instrucción

otro ejemplo son los navegadores chrome, firefox, simulan -interpretan código .html pero no generan salida

pregunta: si yo hago un proyecto de procesamiento de voz (como señales) se te consideraría un compilador porque genera .wav

?

SÍ

podemos hacer lenguaje formal a natural → con reglas de lenguaje formal

Otro ejemplo

Typescript es compilado a JS  
JS se lo pasa a la página web donde se interpreta JS varias veces

### Ejercicio Examen

Encuentra la expresión regular que reconoce cadenas formadas por el alfabeto = {a,b,c,d} pero que no contiene la subcadena bb. aa bb cc dd

$$L(E) = \{E\}$$

$$L(a)$$

aaa  
a\* c

$$L(b)$$

b → bbb bcc

$$L(c)$$

| bbb bbb

$$L(d)$$

| b bbb

bbbbb → aabb bbbbbb

$$a^*(\epsilon + b) + b$$

$$a^* + b^* + c^* + d^*$$

$$\underline{\underline{a^*}} + \underline{\underline{b^*}} + \underline{\underline{b(b)}}^* + \underline{\underline{bb(bb)}}^* + \underline{\underline{c^*}} + \underline{\underline{d^*}}$$

Continuando

Lenguaje formal → compilador → lenguaje natural  
lenguaje C versión formal

Viceversa dependiendo si el lenguaje natural lo ajustamos a versión formal.

GPT-3 - modelo AI  
Traspirar - alto a alto.

programa que simula 1 máquina

compite once

Java slogan (Just in time)

run everywhere

### Máquina Virtual

Interpreta un parte compilada como lo que hace Java ejemplo: Java compila en formato intermedio llamado bytecodes y una máquina virtual los interpreta

Traducen los bytecodes en lenguaje máquina antes de ejecutar el programa intermedio para procesar la entrada.

1 sistema de Cómputo - en el cual (registros ALU, entradas, salidas) canales de silicio encapsulado.

La máquina virtual es 1 sistema de cómputo no real, sin hardware, va simular una máquina real.

Tiempo atrás

Se creó la máquina virtual por 1. Primeros días de internet

Servidor HTML -----> hola.html (GNU/Linux) Java Firefox (MAC)

X no enviar html sino programar "Applets" pequeños mVirtual.exe llegaran a todos los navegadores

JVM.exe se comporta como máquina inventada salida - las arquitecturas eran diferentes (Intel / Sparc) PPC

JVM.out - Linux - dos - MAC .exe - Windows

### 2. Android

se buscaba programar en un lenguaje una app que corriera en Nokia / Huawei / Samsung /

código java .apk viene - - -> .dex2 empaquetado .kotlin

ahora esta ART (Android run time)

esta máquina fue Dalvik VM

Android mandaba la especificación de DVM registros, operaciones en el pasado

Nokia .DVM  
Huawei .DVM  
BOOK



## Presente

no espera que alguien abra x app  
complita desde antes

instala un .apk (Ahead of time compilation)

1. tomo .dex
2. compilo (antes sin decir al usuario)
3. tengo archivo binario .bin de la app listo en la instalación ejecutar

Procesador ARM → Serie J

.class ejecuta en hardware  
sin máquina virtual

Virtual

Dockers? Antes de la máquina virtual (el paso) se le llamaban pailas estas son para los S.O. q no reservan recursos y memoria "independientes" que conectan con sockets para simular servicios virtuales

encarcela el programa y que trabaje an solo conten

## Procesos

los procesos del compilador son 2 grandes bloques

1- análisis

2- síntesis

## análisis

divide el programa

- fuente en componentes e impone una estructura gramatical sobre ellas
- utiliza la estructura para hacer una representación intermedia
- verifica sintaxis o semántica
- trabajo de símbolos

## síntesis

constuye la

traducción (back-end del compilador)

## Fases

flujo de caracteres

### analizador léxico

flujo de tokens

orden correcto  
del lenguaje

### - analizador sintáctico

árbol sintáctico

### analizador semántico

árbol sintáctico decorado

(notas)

### generador código inter.

representación intermedia

### - Optimizador de código independiente de la máquina

representación intermedia

### generador de código

código máquina destino

### Optimizador de código independiente de máquina

código máquina destino

tabla de símbolos.

manejo de errores

## Analizador de léxico

(escaneo) lee el flujo de caracteres que componen el programa fuente y los agrupa en secuencias significativas (lexemas), para c/u lexema produce salida tokens

$\langle \text{nombre\_token}, \text{valor\_atributo} \rangle$

- $\text{símbolo}$   $\text{abstrato}$
- $\text{entrada en la tabla de símbolos}$

ej. asigna al token

operación  $\text{sumaActual} = (\text{valorIni} + \text{valorFin}) * 6$

asignamos  $\langle \text{id}, 1 \rangle$   $\text{etcl, 2} \rangle$

- $\text{símbolo abstracto}$
- $\text{apunta entrada en la tabla de símbolos}$

# ignora los espacios en blanco

$\langle \text{id}, 1 \rangle \langle \text{id}, 2 \rangle \langle + \rangle \langle \text{id}, 3 \rangle \langle * \rangle \langle 6 \rangle$

son símbolos abstractos para operadores.

## tabla de símbolos

1	sumaActual	...
2	valorIni	...
3	valorFin	...

## operación

↓ analizador léxico

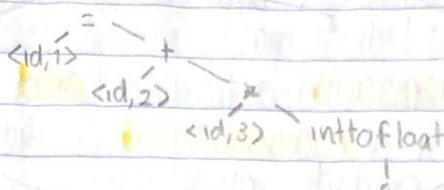
$\langle \text{id}, 1 \rangle \langle \leftrightarrow \rangle \langle \text{id}, 2 \rangle \langle + \rangle \langle \text{id}, 3 \rangle \langle * \rangle \langle 6 \rangle$

↓ analizador sintáctico

$\langle \text{id}, 1 \rangle \langle = \rangle \langle + \rangle \langle \text{id}, 2 \rangle \langle * \rangle \langle \text{id}, 3 \rangle \langle 6 \rangle$

## descripciones de etapas

### análizador semántico



### generador de código intermedio

$$\begin{aligned} t_1 &= \text{inttofloat}(6) \\ t_2 &= \text{id}_3 * t_1 \\ t_3 &= \text{id}_2 + t_2 \\ \text{id}_1 &= t_3 \end{aligned}$$

### optimizador de código

$$\begin{aligned} t_1 &= \text{id}_3 * 6 \\ \text{id}_1 &= \text{id}_2 + t_1 \end{aligned}$$

### generador de código

LDF R2, id3  
MULF R2, R2, #6  
LDF R1, id2  
ADDF R1, R1, R2  
STF id1, R1

## Análisis Sintáctico

(parsing) utiliza los tokens para producir una representación intermedia en forma de árbol - describe la estructura gramatical

Este árbol cada nodo interior son operaciones y los hijos del nodo representan argumentos de la operación, se sigue la regla aritmética de primero hacer la multiplicación y después la suma.

## Análisis Semántico

Utiliza el árbol y la tabla de

símbolos para comprobar la consistencia semántica del programa fuente con la def del lenguaje

comprobación de tipos (float, int, char, etc.)

## generación de código intermedio

puede ser 1 o + representaciones intermedias  
variedad de formas + fácil de optimizar  
esta generación a través del árbol hacen  
similar al código máquina (máquina abstracta)

2 propiedades importantes

- fácil de producir

- fácil de traducir

en el ej. usamos código 3 instrucciones

secuencia de instrucciones similares a  
ensamblador 3 operandos x instr.

c/operando = 1 registro

- 1 operador de lado derecho  
ordenía las operaciones.

- genera nombres temporales.

guarda el valor calculado

## Optimización de código

Busca mejorar el código Intermedio

Se entiende que:

- más rápido

- código mas corto

- código final que consuma menor poder

- menos operaciones - menos bloques de memoria

## Generación de código

le asigna un código final- destino  
generalmente el código máquina  
se selecciona registros & ubicaciones

Asignación judicada de los registros  
para guardar variables

## =Tabla de símbolos=

Debemos de registrar los nombres  
de las variables que se utilizan en el

lenguaje  
nómicos, ensamblador

programa fuente recolec-  
tando información sobre va-  
rios atributos de cada nombre  
Es una estructura de datos  
que tiene registro de c/nombre  
almacenar y obtener datos  
de este registro con rapidez

## Agrupamiento de fases en pasos

Las fases visto tienen que ver con  
la organización lógica de un com-  
pilador. Estas se agrupan en una sola  
pasada, lee el archivo de entrada  
y escribe un archivo de salida

Sabías que el lenguaje  
C es imperativo y también  
lenguaje Von Neumann.

Ventaja de código intermedio

El juego de las fases

lenguaje C → <sup>Código intermedio</sup>  
RTL → binario (core i5)

solo falta RTL → binario  
(logaritmo)

mi propio lenguaje → RTL → me apago  
de GCC

Ingeniería inversa?

el proceso es genérico

↓  
obtengo binario  
para muchas  
plataformas

árbol y pongo plantilla

(operador/ separador) etc.

entonces dado un

ejecutable gracias a

la plantilla quedo con

que compilador lo

generó x su plantilla

dato intercalat



## Sabías que... Reglas Regex

Cuando se dice que Python está hecho en C es porque Py hace la parte del análisis y pone plantillas de C y compilar C es mucho más rápido.

Son patrones utilizados para encontrar una determinada combinación de caracteres dentro de un texto.

### ventajas

manera flexible de reconocer las próximas cadenas de texto que deben recibir con ciertas características.

Las expresiones regulares son extremadamente útiles para extraer información de texto como: códigos, archivos de registro, hojas de cálculo, documentos, etc.

### ejemplos

'hackerrank' - expresión regular  
Welcome to hackerrank, vamos..

Las políticas de (hackerank)

El punto ". " coincide con cualquier carácter excepto

solo letras que no son vocales.

[^aeiou]

con una nueva linea "[^]" negar el conjunto

"[^]" negar el conjunto

cadena inicia con un dígito [0-9], caracteres alfanuméricos [^] punto al final]

^["[ ]"] cualquier carácter debe estar 1 vez dentro de la selección

Buscar apariciones de 1 palabra en un texto, el regex es

regex AProcesar = 'palabraABuscar'

". " significa cualquier carácter para poder evaluar un punto en la expresión regular tenemos "\."

\$ -> evaluar una línea pero solo una macheca posición al final de la cadena

\s -> cualquier carácter en blanco [\r\n\t\f]

\S -> cualquier carácter que no sea espacio en blanco

^ -> macheca posición al inicio de la cadena

\W -> negación de caracteres alfanuméricos

\d -> cualquier dígito [0-9]

\D -> cualquier carácter que no sea un dígito [0-9]

\w -> cualquier carácter de palabra [a-z] [A-Z] [0-9]

alfanuméricos a-z A-Z 0-9

" $\{x\}$ " coincide exactamente con repeticiones de carácter o clase de caracteres o grupos.

" $\{x,y\}$ " hace coincidir en un rango de  $x$  a  $y$  intervalo cerrado

"\*" asterisco coincide con 0 o más repeticiones de 1 char, clase carácter o un grupo

"+" coincide 1 o más repeticiones de un carácter una clase carácter o un grupo.

"\b" afirmar la pos en el límite de una palabra.

3 posiciones

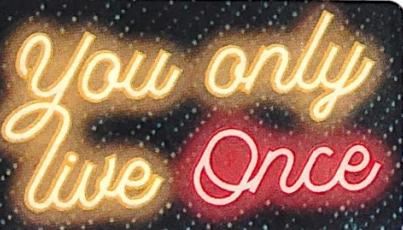
- antes del primer char de la cadena si es char de palabra
- entre 2 char 1 es carácter de palabra y el otro no es char de palabra.
- después del último carácter de cadena si es el último char de palabra.

"(?:)" abando no necesitas que el grupo capture su coincidencia.

"|" barrita - alternancia coincide con solo 1 cosa de varios elementos. Cuando lo usamos dentro de las clases se ajusta a los caracteres.

Si es un grupo coincidirá con expresiones completas derecha o izquierda.

sus parentesis ( )



## Nivel del lenguaje

Los lenguajes tienen niveles

- ① fonológico / gráfico  
la escritura de letras
- ② léxico conozco letras y forma palabras "hola"
- ③ sintáctico forma oraciones "hola, como estas"
- ④ semántico quiere saber como estoy, me saluda.

- ⑤ pragmático qué es la intención detrás de la comunicación.  
- ser cortés  
- estar preocupado

 "Obras de arte" juegas finales para cierto hardware porque los programadores ya conocen el hardware y optimizan al máximo los recursos que tienen

### Ejemplo lenguaje lex.

/\* - separadores  
hola {printf("Hola\n");}  
requer bloque de código

/\* #include <stdio.h> \*/

codigo en C encontrado en bibliotecas

la expresión regular, se ejecuta el código

flex codifica el autómata para la primera parte

- ① ejecuto / edito el .l
- ② compilo en flex bash flex hola.l

- ③ genera lex.yy.c para verlo es vi lex.yy.c

"dentro tiene el autómata que reconoce el código de C" (es un autómata)

- ④ solo compilarlo

gcc lex.yy.c -c

Genera lex.yy.o -o de objetivo

- ⑤ como ensamblarla

gcc lex.yy.c -I -lflex

-lfl -lfl

en otro sistema operativo ej de otra forma

opciones en terminal Ubuntu

- ⑥ ejecuto .a.out

escom  
escom - echo

2021  
2021 - echo

Hola

empata con la expresión regular

Hola

## Opciones

optimización es opcional por la idea / intención del programador

análisis de flujo

quito cosas para hacer algo

así

idea programador

optimización

a=0;

for(int i=0; i<10<sup>6</sup>; i++)

    at++;

printf("%d", a);

puts("10<sup>6</sup>");

porque esto?

- fin -

- quería saber quanto tarda

- quería desbordar

- quería una subrutina y no lo tendrá por esta optimización

quita

for

y solo

saca

valor

Los motores de videojuegos están

hechos a la medida del hardware!

Ojo no se vale optimizar para cosas artesanales like

# Definición de gramáticas

Gramáticas libres de contexto

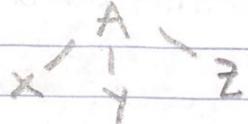
- ① Símbolos terminales - tokens  
son símbolos elementales del lenguaje
- ② Símbolos no terminales - variables  
sintácticas
- ③ producciones en donde cada producción  
consiste en un no terminal (encabezado)  
terminales o no terminales en el cuerpo)
- ④ designación de una de las no  
terminales como el símbolo inicial  
(nombre token - atributos)

## Árbol de análisis sintáctico

Estos árboles muestran de forma gráfica, la manera  
en que el símbolo inicial de  
1 gramática deriva a una  
cadena en el lenguaje.

### Ejemplo

Si el no terminal A  
tiene una producción  
 $A \rightarrow XYZ$  el árbol se  
ve así:



# Parsing

es el encargado del problema de  
tomar una cadena de terminales  
y averiguar como derivarla a partir  
del símbolo inicial de la gramática  
y si no puede derivarse a partir de  
este símbolo, hay que reportar error.

### Propiedades

- a) La raíz se etiqueta con el  
símbolo inicial
- b) cada hoja se etiqueta con  
un terminal o con ε
- c) cada nodo interior se  
etiqueta con un no ter-  
minal
- d) Si A es el no terminal que  
etiqueta a cierto nodo interior  
y  $x_1, x_2, \dots, x_n$  son etiquetas izq.

El analizador léxico saca lexemas que  
pasaran al analizador  
sintáctico para ser  
procesado.

Para esto se hacen  
usos de árboles.



STOP  
AHEAD  
DUDE

# Traducción orientada a la sintaxis

abt  
arb

## atributos

es cualquier cantidad asociada con una construcción de programación  
ejemplo: tipos de dato en expresión  
# de inst. en el código generado, ubicación de primera introducción

## esquemas

de traducción  
es una notación para unir fragmentos de un programa a las producciones de una gramática.

Un árbol sintáctico que muestra los valores de los atributos en cada nodo se le conoce como **árbol de análisis sintáctico anotado**.

## atributos sintetizados

Son cuando su valor en el nodo N de un árbol de análisis sintáctico se determina mediante los valores de los atributos de sus hijos de N y del mismo N.

## Ventajas

- Pueden evaluarse durante un recorrido de abajo hacia arriba

## atributos heredados

Tienen su valor en un nodo de un árbol que se determina mediante los valores de los atributos en el mismo nodo, en su padre y en sus hermanos del árbol.

## Definición orientada a la sintaxis.

La cadena que representa la traducción del no terminal en el encabezado de cada producción es la concatenación de las traducciones de los no terminales en el cuerpo de la producción, en el mismo orden que en la producción con algunas cadenas adicionales entrelazadas.  
Este propiedad se le denomina simde.

## Sabías que...

en la vida real no es necesario construir un árbol de análisis sintáctico **siempre y cuando** aseguremos que las acciones semánticas se realizan como si construyeramos un árbol y después ejecutáramos las acciones durante un **recorrido postorden**:

## Considerar

El analizador sintáctico debe ser capaz de construir el árbol en principio o de lo contrario no se puede garantizar que la traducción sea correcta.

se puede usar  
descenso recursivo o alguna herramienta que genere un traductor directamente de un esquema de traducción

## métodos

Para cualquier gramática libre de contexto, hay un analizador sintáctico que tarda  $O(n^3)$  en analizar una cadena de  $n$  terminales.

En la vida real se hace en complejidad  $O(n)$

## Tipos de analizadores

**descendente:** la construcción de los nodos empieza en la raíz y procede hacia las hojas

**ascendente:** la construcción de los nodos empieza en las hojas y procede hacia la raíz

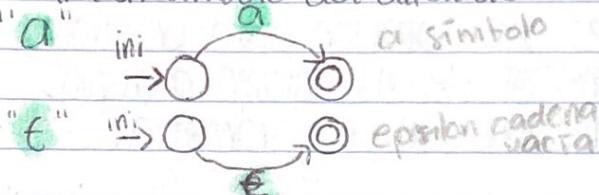
## Gonstrucciones de Thompson

Dada una expresión regular obtenemos un AFN (automata finito no determinista) lo hace a través de plantillas.

\* una plantilla no acepta ningún cambio

## Plantillas

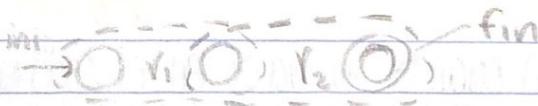
Para un símbolo del alfabeto



dada una expresión regular y se quiere concatenar con otra expresión regular

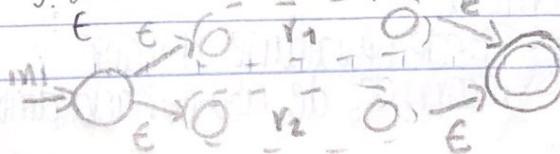
$r_1 \ r_2$

1. Tomamos el automata de la primera expresión
2. Fusionamos el estado inicial  $r_2$  con el estado final  $r_1$



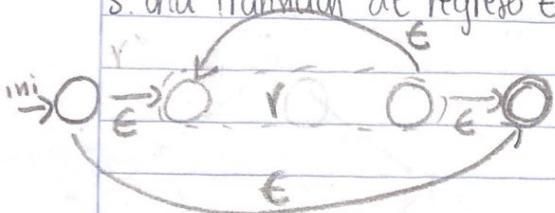
dada una expresión regular  $r_1 \ r_2$

1. Se toma el automata de  $r_2$  abajo de  $r_1$
2. Se crean nuevos estados (inicial y final)
3. Se unen a los 2 caminos con transiciones  $ε$



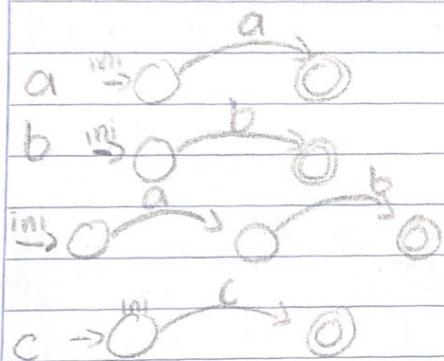
dada la expresión regular  $r^*$

1. Se toma el automata de  $r$
2. Se crea un nuevo (inicial - final)
3. Unimos con transiciones  $\epsilon$  a los nuevos iniciales y final
4. Unimos inicial con final
5. Una transición de regreso  $\epsilon$

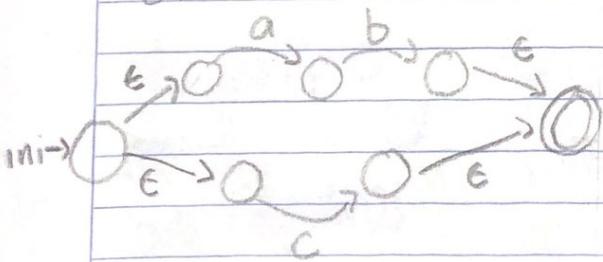


La idea es aplicar las plantillas de forma recursiva  
ejemplo Thompson

$ab|c \rightarrow \text{AFN}$



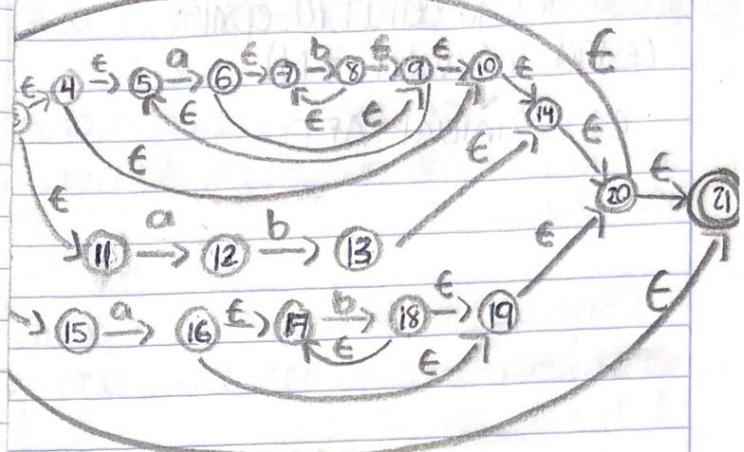
$ab|c$



Ejercicio pasar la sig.  
expresión regular  
 $(abc)^* | b^* c | abc$  a  
un AFN por construcciones de Thompson.

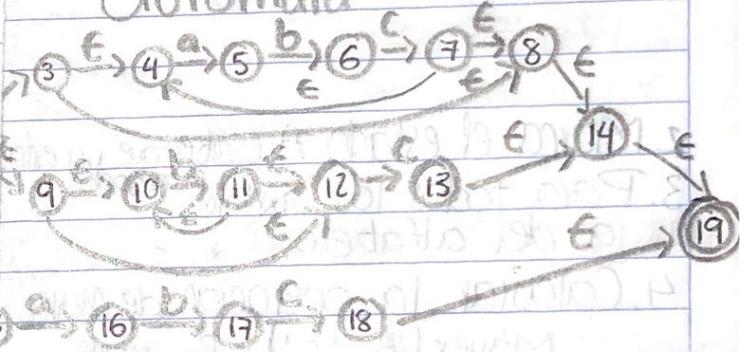
son

### Autómata



construcción 10 min 35 seg  
Thompson

### Autómata



construcción 8 min 07 seg.

para pasar de un AFN (automata no determinista) a AFD (automata finito determinista)

idea

visitar varios estados al mismo tiempo  
dos los estados que se están visitando  
toman un estado del AFD

símbolos.

# Algoritmo

1. Calcular la cerradura-épsilon (estado inicial del AFN)

↓  
estado inicial AFD

retomando el autómata del ejercicio 1

$$\text{cerradura } \bar{\epsilon}(1) = \{1, 2, 3, 4, 5, 10, 11\}$$

$$\cap \text{ transiciones - 1} = 14, 15, 20, 21 \Rightarrow A$$

1 transiciones - 2, 21

2 transiciones - 3, 8, 5

3 transiciones - 4, 11

4 transiciones - 5, 10

5 transiciones - 14

6 transiciones - 20

7 transiciones - 21

2. Marca el estado A (saber que ya está)

3. Para todos los símbolos <sup>procesados</sup> a del alfabeto

4. Calcular la operación de mover

$$\text{Mover}(A, 'a') = B - \text{estado}$$

$$\text{Mover}(A, 'a') = \{6, 12, 16\} = B$$

en 9 con a - no

2 con a - no

3 con a - no

4 con a - no

5 con a - si - 6

10 con a - no

11 con a - 12

14 con a - no

15 con a - si - 16

20 con a - no

21 con a - no

Operación auxiliar

mover(E, a) todos los estados alcanzables con una transición

E: conjunto de estados

a: símbolo del alfabeto

$$P(A, B, C, D, E)$$

5. Calcular la cerradura-épsilon(B)

$$\bar{\epsilon}(B) = \{16, 12, 16\} \cup \{7, 9, 5, 10\}$$

$$\cup \{14, 20, 21, 2, 3, 4, 15, 11, 17, 19\}$$

$$\cup \{C\}$$

$$\cup \{transiciones - 16, 12, 16\}$$

$$\cup \{transiciones - 7, 9, 17, 19\}$$

$$\cup \{transiciones - 10, 20, 5\}$$

$$\cup \{transiciones - 14, 21, 2\}$$

$$\cup \{transiciones - 20, 3, 15\}$$

$$\cup \{transiciones - 21, 4, 11\}$$

$$\cup \{transiciones - 5\}$$

6. Agregar una transición

al AFD  $A - 'a' \rightarrow C$

7. Agregar los estados

nuevos al AFD e ir

al paso 2, por un estado

no marcado

8. Termino cuando no hay

mas estados a procesar

9. Marcar los estados finales

son aquellos que contienen

los estados finales de AFN

estado A será final

estado C será final

## Tabla de referencia

'a': 5 → 6, 11 → 12, 15 → 16  
 'b': 7 → 8, 12 → 13, 17 → 18

Terminando el ejercicio

Paso 1  $C - \epsilon(1) = \{1, 2, 3, 4, 5, 10, 11, 14, 15, 20, 21\} = A$

Paso 2  $(A, a) = \{16, 12, 161, 79, 5, 10, 14, 20, 21, 2, 3, 4, 17, 11, 15, 19\} = B$

Paso 3  $(A, b) = \{\} = C$

Paso 4  $(B, a) = \{16, 12, 16, 7, 9, 5, 10, 14, 20, 21, 2, 3, 4, 17, 11, 15, 19\} = B$

Paso 5  $(B, b) = \{18, 13, 181, 7, 9, 5, 10, 14, 20, 21, 2, 3, 15, 4, 11, 17, 19\} = D$

Paso 6  $(C, a) = \{\} = C$  <sup>núcleo o kernel</sup>

Paso 7  $(C, b) = \{\} = C \rightarrow$  estado de pozo

Paso 8  $(D, a) = \{16, 12, 161, \dots\} = B$

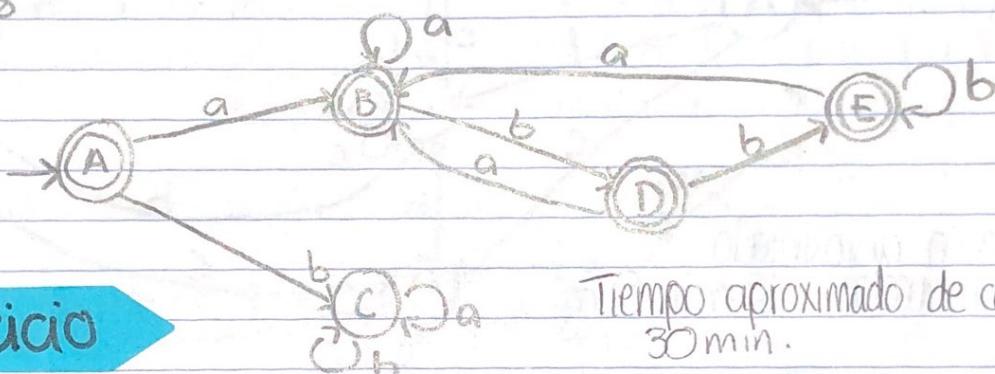
Paso 9  $(D, b) = \{B, 181, 7, 9, 5, 10, 14, 20, 21, 2, 3, 15, 4, 11, 17, 19\} = E$

Paso 10  $(E, a) = \{16, 12, 161, \dots\} = B$

Paso 11  $(E, b) = \{18, 181, \dots\} = E$

Paso 12 terminales: A, B, D, E ya no tengo mas estados a procesar

Dibujo

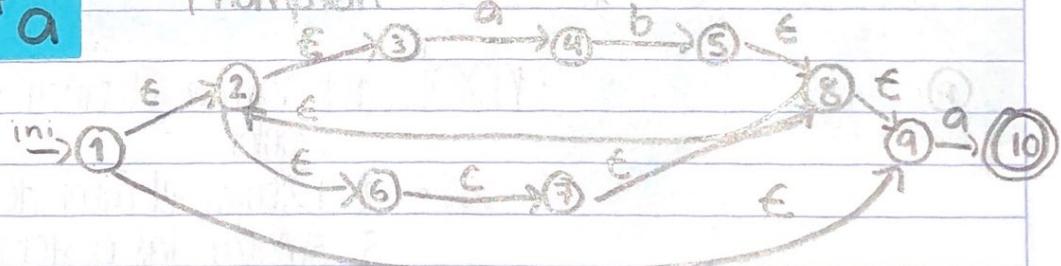


ejercicio

Convertir la ER en AFN (Thompson) AFN a AFD (subconjunto)

$(abc)^* a$

Thompson



Subconjuntos

Tabla de ref.

a: 3 → 4, 9 → 10

b: 4 → 5

c: 6 → 7

Paso 1  $C - \epsilon(1) = \{1, 2, 3, 6, 9\} = A$

2  $(A, a) = \{14, 101\} = B$

3  $(A, b) = \{\} = C$

4  $(A, c) = \{17, 8, 2, 9, 3, 6\} = D$

5  $(B, a) = \{\} = C$

$$6(B, b) = \{15, 8, 9, 2, 3, 6\} = E$$

$$7(B, C) = \{ \dots \} = C$$

$$8(C, a) = \{ \dots \} = C$$

$$9(C, b) = \{ \dots \} = C$$

$$10(C, c) = \{ \dots \} = C$$

$$11(D, a) = \{14, 10\} = B$$

$$12(D, b) = \{ \dots \} = C$$

$$13(D, c) = \{17, \dots \} = D$$

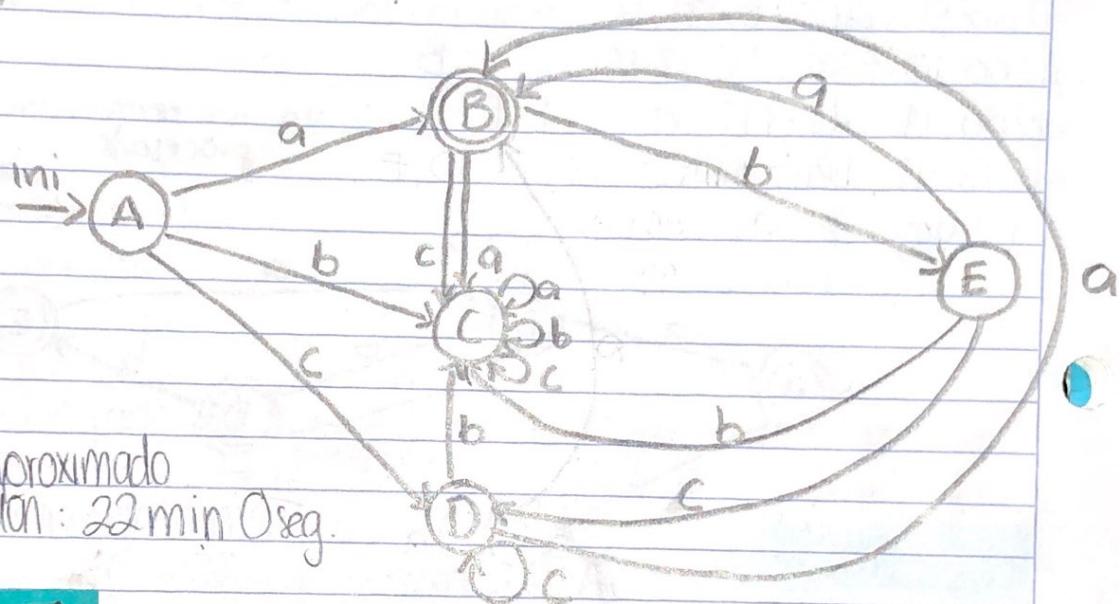
$$14(E, a) = \{14, 10\} = B$$

$$15(E, b) = \{ \dots \} = C$$

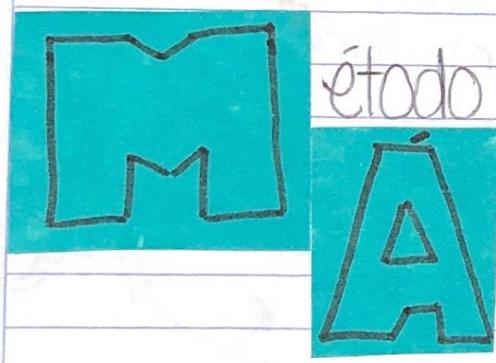
$$16(E, c) = \{17, \dots \} = D$$

$$17 \text{ terminales} = B$$

Dibujo



Tiempo aproximado  
de construcción: 22 min 0 seg.



## éjercicio del árbol

algoritmo

Convierte en un AFD (autómata finito determinista) dada una expresión regular

$$ER \rightarrow AFD$$

1. Marcar el final de la expresión regular
2. Dibujar el árbol de análisis sintáctico
3. Anotar las posiciones de cada símbolo
4. Anotar los nodos anulables
5. Anotar las primeras posiciones de cada nodo
6. Anotar las últimas posiciones de cada nodo
7. llenar la tabla de siguientes.
8. Dibujar el automata

Paso 1 Marca el final

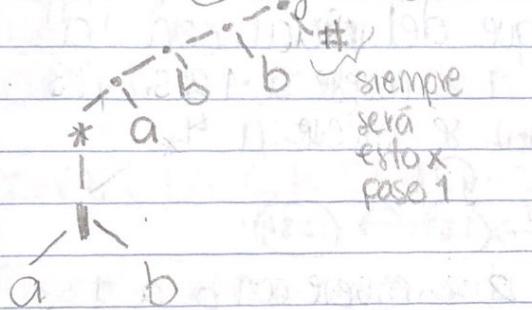
Se usa un símbolo que no pertenezca al alfabeto "#"

se ponen paréntesis a toda la expresión regular, para subir en la precedencia.

Se concatena el símbolo que escogimos "#"

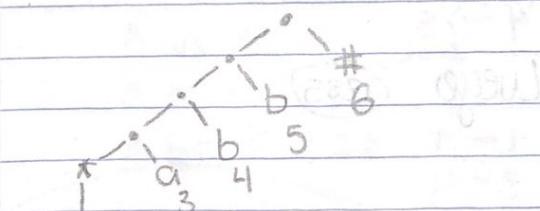
$((a|b)^*abb)^\#$

## Passo 2 Dibujo árbol



### Paso 3: Anotar pos de la ER

$((a \mid b)^* a \mid b b)^*$



á b árbol anotado

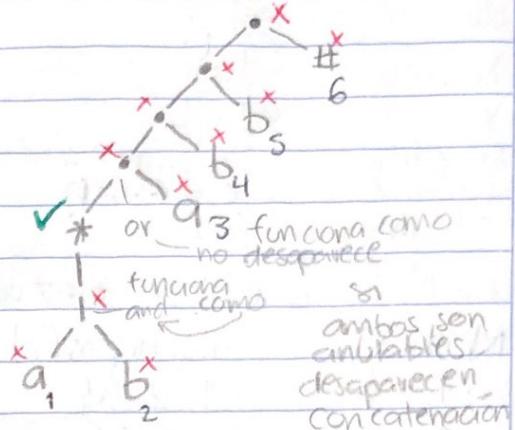
según los símbolos

Paso 4 Anotar nodos anulables (contienen  $\epsilon$  símbolos)

E - es anulable y \* cerradura  
estrella

$$\therefore \underline{(a|b)^*} \\ \text{es anulable}$$

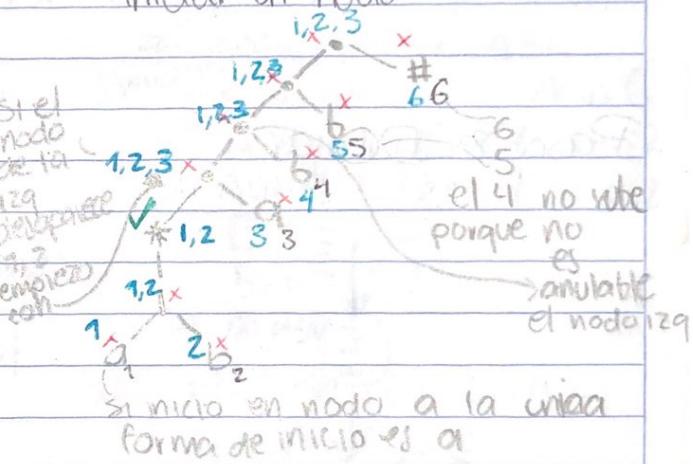
el  $\text{or}(1)$  se anula si alguno de los 2 se anula se quita entonces marcamos).



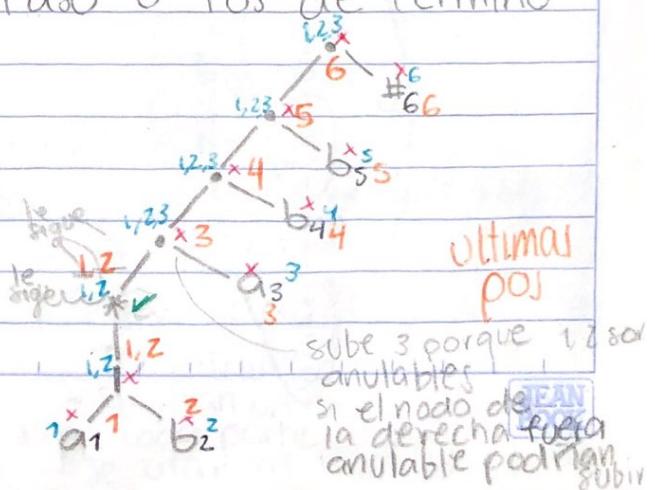
no anulables

✓ anulables

Paso 5 Pos. que pueden iniciar un mdo



rodas arranque  
Paso 6 Pos de término



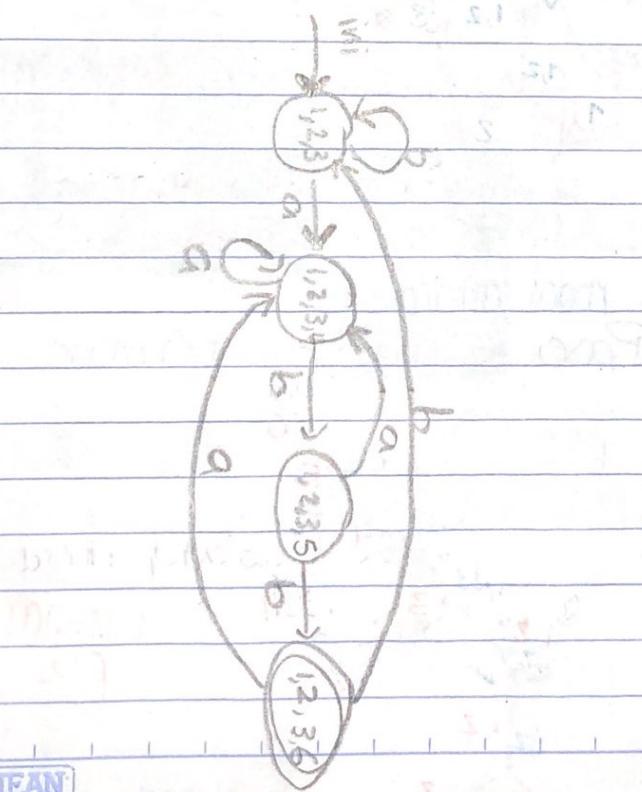
Paso 7 Tabla de las sig. que pos aparece después de cierta pos.

Se calcula visitando el árbol y registrando lugares donde se presenten concatenaciones que se pegan con quien 2 formas las señales y las concatenaciones de iteración, de cerradura

Nodo n siguiente pos (n)

1	le sigue	{1, 2, 3}	
2	le sigue	{1, 2, 3}	concatenación
3	le sigue	{4}	azul
4	le sigue	{5}	azul
5		{6}	último(naranja)
6		Ø	
			azul
			#6
			última naranja
			25

Paso 8 Dibujar



Para dibujarlo el automata arranca con las pos de la raíz {1, 2, 3}

Siguiente estado "a donde se mueve la pos 1"

se mueve {1, 2, 3}

después pos 2? {1, 2, 3}

después pos 3? {4}

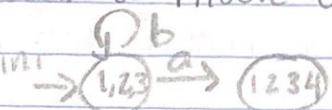
pos 4? {5}

pos 5? {6}

Para la transición tengo que del inicial con 'a'

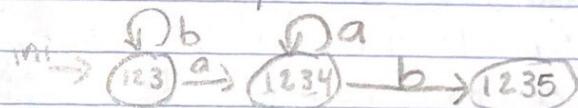
1 se mueve a 123 y 3

2 se mueve a 4



2 se mueve con b a 123

Ahora 123 con a se mueve 1 a 123 y 3 a 4 ∴



2 → 123

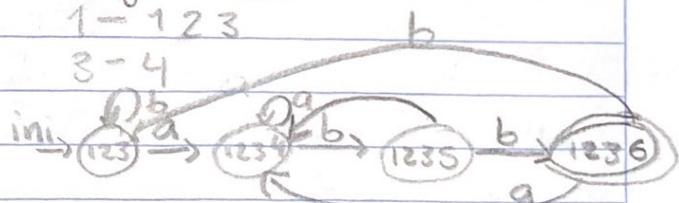
4 → 5

Luego 1235

1 → 123

3 → 4

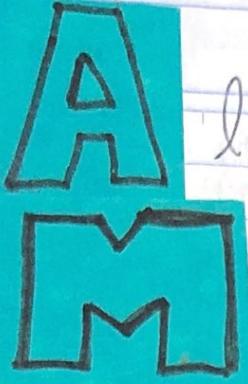
5 → 6



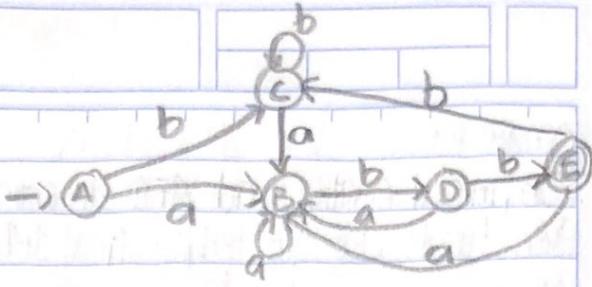
2 va a 123

5 va a 6

quien tiene 6 es final



# Algoritmo de minimización



Va de un AFD a un AFD (con el número mínimo de estados)  
también "algoritmo de las particiones"

A vs D

$D, a \rightarrow B (2)$

$D, b \rightarrow E (1)$   $\Leftrightarrow A \text{ y } D$   
son diferenciables con b

II: 1: {E} 2: {A, B, C} 3: {D}

Paso 1. Crear las particiones

Una con estados finales y otra  
partición con estados no finales.

Paso 3. Repetir paso 2 hasta  
que no pueda dividir

I: {E} {A, C, B, D}

Paso 2. Intentar dividir cada  
partición, buscando estados que  
sean diferenciables.

Intentar dividir: {E}  $\rightarrow$  no se  
puede dividir

Intentar dividir: {A, C, B, D}

A vs C <sup>categoría 2</sup>

$A, a \rightarrow B (2)$

$A, b \rightarrow C (2)$

$C, a \rightarrow B (2)$

$C, b \rightarrow C (2)$

por lo tanto A y C deben  
estar en la misma partición

Intentar partition ABC

A vs B

$A, a \rightarrow (1:2)$

$A, b \rightarrow (1:2)$

$B, a \rightarrow :2$

$B, b \rightarrow :3$

se divide.

III: 1:{E} 2:{D} 3:{A, C}

4:{B}

Si A vs C

$A, a \rightarrow :4$

$A, b \rightarrow :3$

$C, a \rightarrow :4$

$C, b \rightarrow :3$

no se pueden separar

$\therefore$  ya acabe

Paso 4. Dibujar el automá-  
ta mínimo

- Se usan un estado por cada partición
- Se usan las transiciones corres-

II: {E} {2: {A, C}, B, D}

A vs B

$B, a \rightarrow B (2)$

$B, b \rightarrow D (2)$

$\therefore$  no puedo



1. {  
2. #include "ejemplo.tab.h"  
3. } como lo uso  
4. así igual

1. x.  
2. [0-9]+ {return NUMERO;}  
3. [.] {return '.';};  
4. [n] {return '\n';};

fin en flex main(void){

2. flex ejemplo.l yylex();

3. lex.yy.c genera todos los tokens

#include <yyFlex.h>

compilo

4. gcc -c lex.yy.c -o tab.o  
" -c ejemplo.tab.c -o tab.o  
generan .o

Luego los ligo

5. gcc lex.yy.o ejemplo.tab.o -fI  
ejecuto

./a.out desencadenar a  
yyyparse(); bien

función que tiene que llamar  
para arrancar al analizador  
léxico

## Analizador Sintáctico x descenso recursivo

Ejercicio a programar  
gramática

A → aBa

B → bAb

B → C

Con características

cada función tiene que  
hacerse cargo de sus produc-  
ciones y necesita de una  
función auxiliar que consume

### Analizador S. descendente recursivo tablo

La tabulación de estos algorit-  
mos en vez de usar la pila  
de llamadas a funciones, la  
pila será de forma explícita  
ganando profundidad en la  
pila. Están compuestos por 2  
fases ① Construcción de tabla  
② Uso de tabla.  
Existen descendentes y ascendentes  
recuerda que la tabla es el  
automata de pila

## Tipos

1

LL(1) LR RI RR

Left to right (lectura de los tokens)  
L - left most derivation (se busca  
la derivación por la izquierda)

1 - es el # de símbolos de  
anticipación, necesita revisar  
un token para saber que hacer

LL(0) - no necesita revisar tokens

LL(2) - necesita 2 tokens

## análisis diferenciales en gramáticas

Es un método tabular donde primero se construye una tabla, luego se usa la tabla para reconocer cadenas.

Versión no recursiva del A.S.D.R

La tabla codifica el autómata de pila

Cada gramática tiene una tabla (existe una plantilla)

El análisis de cadenas usa la tabla.

Se necesitan 2 funciones auxiliares que son  $\text{first}( )$  -  $\text{follow}( )$

que en español serían  
Primero( ) y Siguientes( )

### Primeros (X) / first

Aquellos símbolos terminales o épsilon que aparecen como primer símbolo de una cadena derivada de X.

### Siguientes (N) / follow

Aquellos símbolos terminales o fin de entrada, que pueden aparecer inmediatamente después de N, durante alguna derivación.

▲ explorar la gramática

### Pilares (U)

Estas funciones son de exploración de gramáticas.

### Reglas para calcular Primeros

Para calcular  $\text{primero}(x)$  para todos los símbolos gramaticales X, aplicamos la siguiente regla

- ① hasta que no puedan agruparse más terminales
- ② o épsilon a ningún conjunto primero

X puede ser terminal o no terminal

Las reglas son

a. Si X es un terminal, entonces  $\text{primero}(x) = \{x\}$

b. Si X es un no terminal,  
 $X \rightarrow Y_1 Y_2 \dots Y_k$  es una producción para cierta  $k \geq 1$ , entonces se coloca a en  $\text{PRIMERO}(x)$ . Si para cierta  $i$ , a está en  $\text{primero}(Y_i)$  y e está en todas las funciones  $\text{Primeros}(Y_i)$ ,  $\text{Primeros}(Y_{i-1})$ ; es decir  $Y_{i-1} \Rightarrow^* e$ .

Si e está en  $\text{primero}(Y_i)$  para todas las  $i = 1, 2, \dots, k$ , se agrega e a  $\text{Primeros}(x)$

c. Si  $X \rightarrow \epsilon$  es una producción entonces se agrega e a  $\text{Primeros}(x)$

d. Podemos calcularlos de una sola vez.

DANGER

## Reglas para calcular siguiente

Se aplican estas reglas hasta

- ① que no pueda agregarse nada a cualquier conjunto siguiente

Las reglas son:

a. Colocar \$ en siguiente(s) en donde S es el símbolo inicial y \$ el delimitador derecho de la entrada.

b. Si hay una producción  $A \rightarrow \alpha B \beta$ , entonces todo lo que hay en primero(β) excepto e está en siguiente(B)

c. Si hay una producción  $A \rightarrow \alpha B \beta$ , una producción  $A \rightarrow \alpha B$  en donde primero(β) contiene a e entonces todo lo que hay en siguiente(A) está en siguiente(B).

gar a  $M[A, a]$ .

② si e está en primero(α) entonces para cada terminal b en siguiente(A), se agrega  $A \rightarrow \alpha$  a  $M[A, b]$ . Si e está en primero(α) y \$ se encuentra en siguiente(A), se agregan  $A \rightarrow \alpha$  a  $M[A, \$]$

Si después de realizar lo anterior, no hay producción  $M[A, a]$  entonces se establece  $M[A, a]$  a error (representado como entrada vacía en la tabla)

$G(S, N, T, P)$

Ejemplo construir tabla LL(1)  
Dada la gramática

$$E \rightarrow TE' \quad (1)$$

$$E' \rightarrow + TE' \quad (2)$$

$$E' \rightarrow \epsilon \quad (3)$$

$$T \rightarrow FT' \quad (4)$$

$$T' \rightarrow * FT' \quad (5)$$

$$T' \rightarrow \epsilon \quad (6)$$

$$F \rightarrow (E) \quad (7)$$

$$F \rightarrow id \quad (8)$$

lenguaje que genera la gramática

$$L(G) = \{ id + id^* id + (id^* id + id) \}$$

$$id + id, id, id^* id, \dots \}$$

\$ → fin de la entrada

## Construcción de Tablas LL(1)

Son concentraciones de información de una G.L.C.

Si la tabla tiene entradas múltiples la G.L.C. no es gramática LL(1)

Si la tabla NO contiene entradas múltiples, es LL(1).

Pasos

Para cada construcción  $A \rightarrow \alpha$  de la gramática

hacer:

- ① Para cada terminal a en primero( $\alpha$ ), agre-

gar a

$$E' \rightarrow +TE'$$

$S(E') = \{+, \$, ;\}$

$$(7) F \rightarrow E$$

$$P('E') = \{'l'\}$$

tengo ciclo, para que tanto es tener otra forma de calcularlo como ya lo hicimos

$$(8) F \rightarrow id$$

$$P(id) = \{id\}$$

Fin de construcción de tabla

$$(4) T = FT'$$

$$P(FT') = \{'l', id\}$$

$$P(F) = \{'l', id\}$$

$$(5) T' = *FT'$$

$$P(T') = \{* \}$$

$$P(*) = \{*\}$$

$$(6) T' \rightarrow E$$

$$S(T') = C \# \text{veces q aparece?}$$

2 veces (4)(5)

Tomo (4)

$$T = FT' \quad A \rightarrow \alpha E'$$

$$S(T) = \{+, \$, ;\}^C$$

'C' # veces q aparece?

2 veces (1)(2)

$$\textcircled{1} \quad S(E) \rightarrow TE' \text{ regla 3}$$

$$P(E') \ (2)(3)$$

$$P(3,1) = \{+, E\}$$

$$S(E) = \{\$, ;\}$$

$$\textcircled{2} \quad E' \rightarrow +TE'$$

$$\text{Tomo (5)} \quad P(E') = \{+, E\}$$

$$T' \rightarrow *FT' \quad S(E') = \{\$, ;\}$$

$$\rightarrow S(T')$$

se cierra

## Pertenencia de L(G)

a) id + id \* id

b) (id)

c) id + \* id

Pertenece o no pertenece usando la tabla



Algoritmo de análisis  
LL(1)

algoritmo de análisis  
sintáctico predictivo.

Establecer ip para que apunte al primer símbolo de w;  
establecer X con el símbolo de la parte superior de la pila;  
mientras ( $X \neq \$$ ) { // la pila no estaría vacía

si (X es a) sacar de pila y avanzo ip;

else si (X es un terminal) error();

else si ( $M[X, a]$  es una entrada de error) error();

else si ( $M[X, a] = X \rightarrow Y_1 Y_2 Y_3 \dots Y_k$ ) {

enviar de salida la producción

$X \rightarrow Y_1 Y_2 \dots Y_k$ ;

sacar de la pila;

meter  $Y_k, Y_{k-1}, \dots, Y_1$  en la pila

con  $Y_1$  en la parte superior;

? establecer X con el símbolo de la parte superior de la pila;

JEAN  
BOOK

## Analizador Sintáctico Ascendente

Parte de la familia LR (LR(0)). Estos analizadores empiezan en las hojas y buscan llegar a la raíz.

L - left to right (lectura)  
R - rightmost derivation  
busca la derivación por la derecha.

0 - no necesita token de anticipación

Sigue siendo método tabular

- ① Se construye tabla
- ② Se usa la tabla

Bison usa un algoritmo de fam LR

Estos métodos utilizan el algoritmo de los subconjuntos - cerradura-e AFN - cerradura-LR mover

Los elementos LR(0)

son producciones de la gramática con un punto indicador

## algoritmo LR

Para construir la tabla se calculan los "subconjuntos LR(0)"

Los subconjuntos LR(0) son agrupaciones de "elementos"

"LR(0)" Algoritmo de los subconjuntos:  
→ subconjuntos de estados

$\{1, 3, 4\} = A$  subconjunto A, contiene estados  
→ funciones auxiliares:  
- cerradura-épsilon()  
- mover()

Dado un subconjunto inicial aplicar cerradura/mover para encontrar otro subconjunto hasta ya no poder agregar más.

## elemento LR(0)

son: producción de la gramática + punto indicador

producción:

$A \rightarrow aBC$   
elementos: indica x lo leí

$A \rightarrow .aBC$

$A \rightarrow a.BC$

$A \rightarrow aB.C$

$A \rightarrow aBC.$

producción:

$A \rightarrow E$

elemento  $A \rightarrow .$

## Construcción de subconjuntos

① Extender la gramática: agregar la producción  $S' \rightarrow S$

$S' \rightarrow S \leftarrow$   
 $S \rightarrow aBCa$   
 $S \rightarrow bCA$   
 $S \rightarrow BC$   
el ascenso a la raíz es único.

② Aplicar la cerradura ( $\{S'\} \rightarrow \cdot S\}$ ) = primer subconjunto

③ Para todos los símbolos

x de NoTerminales

3.1 Cerradura(Movr(C, x)) = nuevo subconjunto.

3.2 Agregar a los subconjuntos

## Familia LR

se les llama

así porque tienen la misma plantilla (tabla, conjuntos, pila)

saber si una cadena pertenece

LR(0) - ya

SLR - su forma sintetizada.

LR(1) → tiene producción punto token

LALR → su forma sintetizada

(ventajas pueden ver mas errores)

GLR → caminos genéricos

PLR → como otro aparte si hay múltiples entradas, no son LR

Para los errores existen varias técnicas, tokens de error gramática de error, etc.

## Ejemplo

Construir los subconjuntos

LR(0)

$E \rightarrow E + T$

$E \rightarrow T$

$T \rightarrow T^* F$

$T \rightarrow F$

$F \rightarrow (E)$

$F \rightarrow id$

Calcular el subconjunto inicial

cerradura( $\{E'\} \rightarrow \cdot E\}$ ) =

$\{ | E' \rightarrow \cdot E |, E \rightarrow \cdot E + T, E \rightarrow \cdot T, T \rightarrow T^* F,$   
 $T \rightarrow \cdot F, F \rightarrow \cdot (E) |, F \rightarrow \cdot id \} \quad (1)$

ciclo para todos los  $x (T, N)$

movr((1), E) =  $\{ | E' \rightarrow E \cdot, E \rightarrow E \cdot + T | \} \quad (2)$

cerradura

$CIM((1), T) = \{ | E \rightarrow T \cdot, T \rightarrow T^* F \} \quad (3)$

$(1), F = \{ | T \rightarrow F \cdot | \} \quad (4)$

$(1), () = \{ | F \rightarrow ( \cdot, E ) |, E \rightarrow \cdot E + T,$

$E \rightarrow \cdot T |, T \rightarrow \cdot T^* F, T \rightarrow \cdot F,$   
 $F \rightarrow \cdot ( E ), F \rightarrow \cdot id \} \quad (5)$  aceptación

$(1), id = \{ | F \rightarrow id \cdot | \} \quad (6)$

$(2), + = \{ | E \rightarrow E \cdot + T |, T \rightarrow T^* F,$   
 $T \rightarrow \cdot F, F \rightarrow \cdot ( E ), F \rightarrow \cdot id \} \quad (7)$

$(3), * = \{ | T \rightarrow T^* \cdot F |, F \rightarrow \cdot ( E ),$   
 $F \rightarrow \cdot id \} \quad (8)$

$(5), E = \{ | F \rightarrow ( E \cdot ), E \rightarrow E \cdot + T | \}$

$(5), T = (3), (5), () = (5) \quad (9)$

$(5), F = (4) \quad (5), id = (6)$

MAN BOOK